CHAPITRE X

LE PASSAGE PAR NECESSITE ET LES FLUX

Possibilité déjà réalisée en Lisp et Scheme, en Caml, les "streams", "flots", "flux" ou "segments" ne sont pas évalués quand ils sont construits. Ils sont délimités par [<...; ...; .... >], une apostrophe indique que l'on prend une valeur et non un flux (d'évaluation différée). Les flux sont les seuls objets de Caml soumis à une évaluation paresseuse.

```
let sc = [<'`a`; '`b`; '`c` >];;        ☞ sc : char stream = <abstract>
let si = [<'1; '2; '3; '4; '5 >];;      ☞ si : int stream = <abstract>
```

On considère par exemple le flux débutant par 0 suivi de "si" lequel est encore suivi de "si", et qui se termine par 9 :

```
let S = [<'0; si; si; '9 >];;           ☞ S : int stream = <abstract>
```

On dispose de quelques fonctions prédéfinies stream_next, qui donne le premier élément (mais en avançant, attention la lecture est destructive), stream_get, qui donne le couple formé du premier élément et du flux suivant, mais qui n'est pas destructif, et stream_of_string :

```
let si = [<'1; '2; '3; '4; '5 >];;      ☞ si : int stream = <abstract>
stream_next si;;                        ☞ int = 1
stream_next si;;                        ☞ int = 2
stream_next si;;                        ☞ int = 3
stream_get si;;                         ☞ int * int stream = 4, <abstract>
stream_get si;;                         ☞ int * int stream = 4, <abstract>
stream_next si;;                        ☞ int = 4
stream_next si;;                        ☞ int = 5
        (* renvoie une erreur si on l'appelle encore une fois *)
```

Pour obtenir le calcul du n-ième élément à partir de 1 cette fois ("fun" n'est pas accepté).

1. Obtention du n-ième élément

Dans le filtrage qui suit, une écriture match f with [<'a; q>] peut être remplacée par match f with [<'a>] puisque dans les deux cas la lecture va modifier f, et au lieu de noter q, ce qui suit l'élément a lors de l'évaluation, on peut reprendre le paramètre f qui désigne alors la même chose.
Comme le nom l'indique, un flux de données est lu, et donc, à chaque instant se trouve en une position de lecture (attention, l'accès est destructif).
Ce qui fait qu'après la question précedente, si se trouve au quatrième rang, et donc "nth 2 si" donnerait 5.

```
let rec nth n = function [< 'a; f >] -> (if n = 1 then a else nth (n - 1) f);;
```

```
nth 3 si;; ☞ int = 3
```

Pour avoir la liste commençante des n premiers éléments sur le flux f :

```
let rec debut n = function [< 'a; f >] -> (if n = 0 then [] else a :: (debut (n - 1) f))| [<>] -> [] ;;
    ☞ debut : int -> 'a stream -> 'a list = <fun>
```

```
debut 3 (stream_of_string "abcdef");;   ☞ char list = [`a`; `b`; `c`]
```

2. Un map sur les flux

Remarque importante, fabriquer l'analogue d'un "map" ne peut se dispenser de faire appel à l'apostrophe pour bien signaler qu'on souhaite les valeurs rendues par f appliquées aux valeurs du flux, (une fonction prédéfinie "do_stream" existe, mais renvoie unit) sinon
"let rec map f = function [< 'x; r >] -> [< f x; map f r >];;"
serait du type ('a -> 'b stream) -> 'a stream -> 'b stream, aussi écrit-on :

```
let rec mapflux f = function [< 'x; r >] -> [< '(f x); mapflux f r >];;
    ☞  ('a -> 'b) -> 'a stream -> 'b stream
```

```
let si = [<'1; '2; '3; '4; '5 >];;          ☞ si : int stream = <abstract>
debut 3 (mapflux (fun x -> (x + 2)) si);;   ☞ int list = [3; 4; 5]
```

3. Filtre

Construisons à présent une fonction lisant un flux et produisant le flux des éléments vérifiant une propriété p :

```
let rec filtre p fl = match fl with
    | [< 'x>] -> if p x then [< 'x; (filtre p fl) >] else [< (filtre p fl)>]
    | [<>] -> [<>];;
```

Le fait de construire ci-dessous "sip" à partir de "si" puis de demander son premier entier, non seulement avance dans "sip" mais dans "si" :

```
let si = [<'1; '2; '3; '4; '5 >] and sip = filtre (fun x- > x = 2 * (x / 2)) si;;
stream_next sip;; ☞ 2
stream_next si;; ☞ 3
```

Alors que :

```
let si = [<'1; '2; '3; '4; '5 >] and sip = filtre (fun x-> x = 2 * (x / 2)) si;;
stream_next si;; ☞ 2
stream_next sip;; ☞ 4
```

4. Construction de suites infinies

On peut manipuler des objets infinis grâce aux flux, ainsi :

```
let rec entiersup n = [< 'n; (entiersup (n + 1)) >];;
```

```
let N = entiersup 0;;
```

```
stream_next N;; ☞ 0
stream_next N;; ☞ 1
stream_next N;; ☞ 2
stream_next N;; ☞ 3
stream_next N;; ☞ 4
stream_next N;; ☞ 5
```

Alors que la vérification de type serait bonne, mais la construction (qui ne se fait pas à la déclaration) serait impossible dans la formulation :

let N = [< '0; 'stream_next N'; snd(stream_get N') >]
where rec N' = [<'succ(stream_next N); snd(stream_get N) >];;

Sur le même modèle on peut construire toute suite définie par récurrence, par exemple Fibonacci.

```
let rec fib n m = [<'n; (fib m (n+m)) >];;
```

```
let f = fib 1 1;;
stream_next f;;☞ 1
stream_next f;;☞ 1
stream_next f;;☞ 2
stream_next f;;☞ 3
stream_next f;;☞ 5
stream_next f;;☞ 8
stream_next f;;☞ 13
stream_next f;;☞ 21
```

5. Flux somme de flux

```
let rec som f f' = match f with
        | [< 'a >] -> (match f' with | [< 'b >] -> [< '(a + b); (som f f')>]
                                     | [<>] -> [<>])
        | [<>] -> [<>];;
```

☞ som : int stream -> int stream -> int stream = <fun>

```
let si = [<'1; '2; '3; '4; '5 >];;      ☞ si : int stream = <abstract>
let si' = [<'7; '4; '1 >];;             ☞ si' : int stream = <abstract>
stream_get (som si si');;               ☞ int * int stream = 8, <abstract>
stream_next (som si si');;              ☞ int = 6       (* et ainsi de suite *)
```

En ML, on aurait soit une définition de doublet comme suit :
#type 'a flux = lazy doublet of 'a * 'a flux, soit encore plus simple, comme en Lisp, une directive d'évaluation "paresseuse" (avec le terme prédéfini "df") permettant des listes :
"let rec fib = doublet (1, fib') and fib' = doublet(1, (som fib fib'));;" ou bien :
"let rec fib = 1 :: fib' and fib' = 1 :: (som fib fib');;" nth 5 fib;; qui devrait donner 8.

6. Application au crible d'Eratosthène

En concevant une fonction "crible" qui, appliquée sur un flux d'entiers, va avancer dans ce flux en éliminant les multiples de l'entier lu dans la suite, et ceci indéfiniement, on va pouvoir construire la suite des nombres premiers :

```
let nondiv a b = (b mod a) <> 0;;
```

```
let rec crible fl = match fl with   [<'n>] -> [< 'n; (crible (filtre (nondiv n) fl)) >]
                                    | (* inutile *) [<>] -> [<>];;

let premiers = crible (entiersup 2);;
```

```
stream_next premiers;; ☞ 2
stream_next premiers;; ☞ 3
stream_next premiers;; ☞ 5
stream_next premiers;; ☞ 7
stream_next premiers;; ☞ 11
stream_next premiers;; ☞ 13
stream_next premiers;; ☞ 17
stream_next premiers;; ☞ 19
stream_next premiers;; ☞ 23
```

```
let prem n = let fp = crible (entiersup 2) in
        for i = 1 to n do print_int (stream_next fp); print_string " " done;;
```

```
prem 20;; ☞  2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71
```

```
let rec listprem n  = fst (aux n (crible (entiersup 2)))
        where  rec aux k fl = if k = 0 then ([], fl)
                        else match fl with [<'x>] -> let q, ff = aux (k - 1) fl in (x :: q, ff);;
```

```
listprem 23;;
☞  [2; 3; 5; 7; 11; 13; 17; 19; 23; 29; 31; 37; 41; 43; 47; 53; 59; 61; 67; 71; 73; 79; 83]
```

Rappelons qu'avec la syntaxe très épurée de Haskell, la définition "à la Prolog" par clauses et le fait que l'évaluation paresseuse y est la règle, la même idée peut être mise en oeuvre en quatre lignes avec des listes :

```
entiersup n = n : entiersup (n+1)
retmult n (x : q) | x mod n == 0 = retmult x q | otherwise = x : (retmult x q)
crible (x : q) = x : crible (retmult x q)
premiers = crible (entiersup 2)
```

CHAPITRE XI

LE HASARD

ALGORITHMES D'EVOLUTION

Le hasard

Disposant des fonctions "rand__int" qui fournit un entier entre 0 inclus et son argument exclus, (et de son équivalent random__float r;;) il est possible de simuler le hasard.
Pour initialiser une nouvelle suite aléatoire, le meilleur moyen est encore d'attendre la pression d'une touche, car la durée d'attente sera toujours différente :

```
let s = ref 0 in while not (key_pressed ()) do incr s done; read_key (); random__init !s ;;
```

1. Trouver un élément au hasard entre deux nombres ou dans une liste abstraite

On peut construire une fonction utile "has" donnant un élément au hasard entre deux entiers a et b inclus. "hasard" pour donner un élément d'une liste non vide q :

```
let has a b = a + random__int (b -  a + 1);; (* renvoie un entier entre les entiers a et b inclus *)

let rec nth (a :: q) = fun 0 -> a | n -> nth q (n  -  1)          (*élément n de liste à partir de 0*)
and hasard q = nth q (random__int (list_length q));;
```

2. Simulation d'une loi de Poisson ou de Gauss

On simule une loi normale de Gauss, en tirant au hasard, de façon uniforme (on vérifie que "random__int" l'est à peu près) un grand nombre de fois.

```
let gauss m s = let som = ref 0 in for i = 1 to 12 do som := !som + random__int 100; done;
                s *. (float_of_int (m - 6) +. (float_of_int !som) /. 100.);;
```

```
for i = 1 to 100 do print_string ((string_of_float (gauss 0 0.01))^" ") done;; (* vérification *)
```

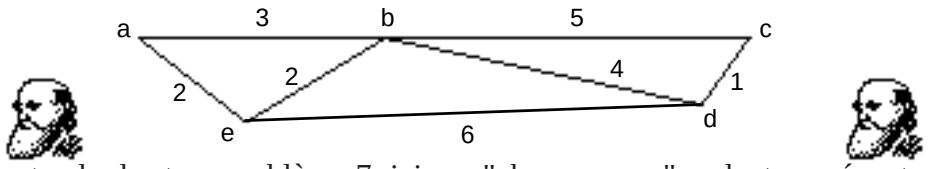
☞ -0.0039 -0.0117 0.012 0.019 -0.002 -0.0015 -0.0052 -0.0067 0.0098 -0.0066 0.0081 0.0074 0.0142 0.0022 0.003 0.0059 -0.0007 0.0031 0.0095 -0.0044 0.0102 0.0096 -0.0174 0.0082 0.01 0.0002 0.0095 0.0124 0.0111 0.0018 -0.0012 0.0195 -0.0076 -0.0122 -0.0043 0.0049 0.008 -0.0006 0.0052 -0.0121 -0.0157 0.0056 -0.0184 0.0132 -0.0101 -0.0037 0.0143 -0.0096 -0.0131 -0.0019 -0.0095 -0.0035 -0.0008 0.0086 0.0137 0.0016 -0.0059 -0.0107 -0.0074 0.0201 -0.0014 0.006 -0.007 0.0016 0.0063 -0.0168 0.0028 -0.0012 0.004 -0.0164 -0.0035 -0.0067 -0.0058 ... - : unit = ()

```
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.));;
```

```
for i = 1 to 100 do print_string ((string_of_int (poisson 3))^" ") done;; (* Pour vérification *)
☞  1 5 3 2 3 2 1 3 2 6 2 3 1 2 4 1 3 2 3 5 4 0 7 5 3 4 6 5 3 5 0 2 1 0 3 5 0 1 5 4 2 1 3 5
1 2 2 2 4 0 5 1 6 6 2 4 0 5 2 2 1 3 3 5 5 5 4 3 5 4 7 3 5 7 1 4 5 2 2 5 2 5 2 2 6 3 2 4 3 1 2 7
4 4 6 0 3 5 0 3 - : unit = ()
```

3. Exemple des algorithmes génétiques

Un problème d'optimisation simple à définir est celui du voyageur de commerce : minimiser la distance avec la contrainte de visiter chaque ville. Par exemple dans le petit schéma, une solution sera un mot formé sur l'alphabet A = {a, b, c, d} et sa performance sera la somme des distances de l'itinéraire avec un malus m = 9 (par exemple) par ville manquante ou par répétition.



L'algorithme standard est au problème 7, ici, un "chromosome" codant une éventuelle solution du problème sera un article formé par la liste des villes et la distance correspondante :

```
type chromosome = {valeur : int; route : string list};;
let m = 9 and al = ["a"; "b"; "c"; "d"; "e"];;
        (* malus = 9 et alphabet des "gènes" pouvant former un chromosome *)

let rec dist = fun "a" "b" -> 3 | "a" "c" -> 8 | "a" "d" -> 7 | "a" "e" -> 2 | "b" "c" -> 5
                   | "b" "d" -> 4 | "b" "e" -> 2 | "c" "d" -> 1 | "c" "e" -> 7 | "d" "e" -> 6
                   | x y -> if x = y then m else dist y x;;

let rec ret x = fun [] -> [] | (a :: q) -> (if x = a then q else a :: (ret x q));;
let rec tete q n = if q = [] or n = 0 then [] else (hd q) :: (tete (tl q) (n - 1)) (* n premiers éléments  *)
and queue q n = if q = [] or n = 0 then q else queue (tl q) (n - 1);; (* tout sauf les n premiers de q *)
let segment p q ll = tete (queue ll p) (q - p + 1);; (* des rangs p à q inclus de ll *)
let rec implose = fun [] -> ""|  (a :: q) -> a ^ (implose q);;
```

Les opérateurs génétiques mutation, migration, transposition, croisement consistent à modifier au hasard une liste de villes (les fonctions nth, has, hasard, poisson étant reprises :

```
let mut lv = (* liste de villes -> liste de villes *)
let n = random__int (list_length lv) and g = hasard al in (tete lv n) @ (g:: (queue lv (n+1)));;

let mig lv = (* tout est perturbé sauf la première ville qui passe en dernier *)
        let e = ref [hd lv] and n = poisson 4 in for i = 1 to n do e := (hasard al)::(!e) done; !e;;

let transpo lv = let p = random__int ((list_length lv) / 2) in
        let q = has (p+1) ((list_length lv)-1) in
        (tete lv p) @ (((nth lv q) :: (segment (p+1) (q-1) lv)) @ ((nth lv p) :: (queue lv (q+1))));;

let crossover lv1 lv2 = let l1 = list_length lv1 and l2 = list_length lv2
        in let p = random__int ((min l1 l2) / 2) in let q = has (p+1) (min l1 l2) in
        (tete lv1 p) @ ((segment p q lv2) @ (queue lv1 (q + 1))),
        (tete lv2 p) @ ((segment p q lv1) @ (queue lv2 (q + 1)));;
```

```
mut ["e"; "d"; "f"; "r"; "g"];;
☞ string list = ["e"; "d"; "d"; "r"; "g"]
mig ["e"; "d"; "f"];;
☞ string list = ["b"; "c"; "b"; "c"; "b"; "e"; "e"]
transpo [1;2;3;4;5;6;7;8;9];;
☞ int list = [1; 2; 3; 7; 5; 6; 4; 8; 9]
crossover [1;2;3;4;5;6;7;8;9] [10;11;12;13;14;15];;
☞ [1;2;12;13;14;6;7;8;9], [10;11;3;4;5;15]
```

On choisit une stratégie d'évolution simple, où chaque chromosome de la population est comparée avec un "voisinage" formé par ses enfants en appliquant tous les opérateurs choisis. Le meilleur d'entre eux est conservé pour la génération suivante, la population n'est pas triée.

```
let rec produc n = if n = 0 then [] else (mig ["a"]) :: (produc (n - 1));;
let val lv = {valeur = (malus al lv) + (somd (hd lv) lv); route = lv}
        where rec malus  lr = fun [] -> m * (list_length lr) | (x :: lv) -> malus (ret x lr) lv
        and somd v = fun [x] -> dist x v | (x :: y :: lv) -> (dist x y) + somd v (y :: lv);;

    val ["a"; "b"; "d"; "e"];;
    ☞ chromosome = {valeur=24; route=["a";"b";"d";"e"]} *)

let valoriser p = map val p;; (* calcule la valeur des individus d'une population *)

let meilleur e = meilleur' (hd e) (tl e) (* renvoie le meilleur ind. d'une population e *)
        where rec meilleur' I = fun [] -> I | (i :: lc) -> meilleur' (if I.valeur < i.valeur then I else i) lc;;

let generation p = generation' [] p (* lv = liste vue, i = indice courant, lr = liste restante *)
        where rec generation' lv = fun [] -> lv | (i ::  lr) -> let r = i.route in
        generation' ((meilleur (i :: (valoriser [mut r; mig r; transpo r; f1; f2]))) :: lv) lr
where (f1, f2) = crossover r (hasard (if list_length lv < list_length lr then lr else lv)).route ;;
```

L'évolution est lancée avec un nombre de np d'individus initialement aléatoires, et un nombre fixé ng de générations.

```
let evol np ng = let p = ref (valoriser (produc np)) in
        for g = 1 to ng do p := generation !p; let i = meilleur !p in
                print_string ("Génération " ^ (string_of_int g) ^ " : meilleur itinéraire " ^
                (implose i.route) ^ " de distance " ^ (string_of_int i.valeur) ^ "\n") done ;;

    evol 2 2;;
    ☞ Génération 1 : meilleur itinéraire ea de distance 31
    Génération 2 : meilleur itinéraire eabdc de distance 17
```

C'est très rapide, mais le problème était ici de taille très réduite.

4. Théorie des équilibres ponctués

On considère un individu ancêtre constiué d'un chromosome représenté par g gènes (des lettres) donnant naissance à une population de np individus. A chaque génération, tous les individus sont remplacés deux par deux par leurs croisements sur lesquels s'exercent une mutation simple. On calcule l'appariement moyen f avec l'ancêtre, de la population supposée de taille constante np, au bout de ng générations. A priori, la fonction ng -> f doit être globalement décroissante vers 0. On cherche à vérifier si cette décroissance est ralentie pour une faible taille, une population ayant d'autant plus de chances d'être conforme à l'ancêtre, qu'elle est réduite. On reprend nth, has, hasard, ret, tete, queue, segment, crossover.

```
let ancetre g = let i=ref [] in for k=96+g downto 97 do i:=(char_of_int k) :: (!i) done; !i;;
let rec repete n i = if n = 0 then [] else i :: (repete (n - 1) i);;
let vue g np ng = let a = ancetre g in (let p = ref (repete np a) in for k = 1 to ng do
                p := generation !p []; print_int (frequence !p); print_string "/1000 "done)
        where rec generation p r = if p = [] then r else (let i = hasard (tl p) in
                let e1, e2 = crossover i (hd p) in generation (ret i (tl p)) ((mut e1) :: (mut e2) :: r))
        and frequence p = let f = ref 0 in for i = 0 to np - 1 do for j = 0 to g - 1 do
                if nth (nth p i) j = char_of_int (97 + j) then incr f done done; (!f * 1000) / (np * g)
        and mut lv = let n = random__int g in (tete lv n) @ ((hasard a) :: (queue lv (n + 1)));;
```

En faisant un grand nombre d'appel pour un même nombre de gènes g et de générations ng (plus de 50 car la décroissance peut s'avérer peu régulière), on observe bien l'hypothèse annoncée. Par

exemple pour 10 gènes, la fréquence 0,5 est obtenue en moyenne après 10 générations si np de l'ordre de 10, et vers la septième si np de l'ordre de 50.
Par ailleurs, ce qui est conforme aux probabilités, cette décroissance est d'autant plus ralentie que le nombre de gènes est élevé (les espèces les moins évoluées (en terme de complexité du génome) pourraient donc diverger plus rapidement).

5. Les dames de Gauss par un algorithme génétique

On donne l'algorithme d'optimisation "steady-state genetic algorithm" SSGA ($\mu = 12$, $\tau = 3$) destiné à trouver une solution dont l'évaluation f est minimale : μ solutions aléatoires sont initialement produites comme chaînes comportant n symboles (gènes). Cette population est évaluée et triée. De génération en génération chaque "solution" de la population courante produit un enfant (au hasard) soit par la mutation d'un de ses gènes, soit par croisement à un site avec un autre parent. La population fille est systématiquement formée du tri des $\mu - \tau$ meilleurs parents et des τ meilleurs enfants. Suivant le problème, l'arrêt du processus d'évolution peut être contraint par un nombre de générations ou d'évaluations de f, ou par une valeur attendue de f.

Pour le problème des reines de Gauss, f sera le nombre de couples de reines en prises. Il faut donc produire des "solutions" au hasard (fonction "chrom") et trouver une solution donnant le minimum 0. La fonction "evaluer" est à minimiser. Cette fonction prend une solution comme liste de huit numéros de lignes si n = 8 par exemple, et lui rajoute en tête, son nombre de couples en prises.

```
let rec compte n r = if n=0 then r else compte (n-1) (n :: r);;
let rec tri = fun [] -> [] | [x] -> [x] | (p :: q) -> sep p [] [] q
and sep p pp pg = fun [] -> (tri pp) @ (p :: (tri pg)) (* c'est le tri par pivot *)
        | ((x :: q) :: lr) -> if x < hd p then sep p ((x :: q) :: pp) pg lr else sep p pp ((x :: q) :: pg) lr;;

let rec nth q n = if n = 0 then hd q else nth (tl q) (n-1);;
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);; (* q sauf ses n premiers *)
let hasard q = nth q (random__int (list_length q)) and has a b = a + random__int (b - a + 1);;
let rec chrom ng r n = if ng=0 then r else chrom (ng - 1) ((has 1 n) :: r) n;;
let evaluer c = (gauss 0 c) :: c
        where rec gauss k = fun [] -> k | (x :: col) -> gauss (k + (prises x 1 0 col)) col
        and prises x p k = fun [] -> k | (y :: col) -> if mem y [x; x + p; x - p] then
                                        prises x (p + 1) (k + 1) col else prises x (p  + 1) k col;;
```

```
Exemple          compte 7 [];; ☞ [1; 2; 3; 4; 5; 6; 7]
tri [[8; 4; 2]; [5; 3]; [0; 1; 2; 3]; [4; 0]; [3; 6; 9]];;
☞ [[0; 1; 2; 3]; [3; 6; 9]; [4; 0]; [5; 3]; [8; 4; 2]]
chrom 12 [] 4;; ☞ [3; 1; 4; 3; 3; 3; 1; 3; 3; 3; 3; 2]
evaluer [5; 7; 5; 2; 3; 4; 2; 1];; ☞ [10; 5; 7; 5; 2; 3; 4; 2; 1]
```

```
let rec init mu n r = if mu = 0 then tri (map evaluer r) else init (mu-1) n ((chrom n [] n) :: r);;
let cross p k m = (tete p k) @ (queue m k)
and mut c k n = (tete c (k - 1)) @ ((has 1 n) :: (queue c k));;
```

```
init 9 5 [];; ☞ [[3; 3; 3; 5; 4; 2]; [3; 3; 3; 5; 2; 5];  [3; 5; 3; 5; 3; 2]; [4; 1; 5; 2; 3; 4]; [4; 5; 1;
2; 5; 5]; [5; 3; 2; 2; 4; 5]; [5; 2; 3; 1; 3; 3]; [5; 3; 4; 4; 3; 1]; [8; 3; 4; 3; 3; 3]]
cross [9; 8; 7; 6; 5; 4] 3 [0; 1; 2; 3; 4; 5];; ☞ [9; 8; 7; 3; 4; 5]
mut [9; 8; 7; 6; 5; 4] 4 9;; ☞  [9; 8; 7; 9; 5; 4]
```

La fonction "init" permet de produire une liste de mu "listes-solutions" de longueur n, qui vont être triées de façon croissante suivant leur évaluation "evaluer".
Puis la fonction "generation" permet d'engendrer la population suivante en tronquant la génération précédente, ainsi que celle des "enfants" qui ont été évalués et triés.

```
let rec enfants f n = fun [] -> tri (map evaluer f)
        | (p :: pr) -> enfants ((if has 0 9 < 5 then mut p (has 0 n) n
                                 else cross p (has 0 n) (* soit mutation, soit croisement appliqué à p *)
                                                          (hasard (if pr = [] then f else pr))) :: f) n pr;;
```

```
enfants [] 5 [[1; 2; 3; 4; 5]; [2; 1; 2; 1; 2]; [3; 4; 1; 2; 5]; [5; 4; 5; 3; 1]];;
☞ [[2; 2; 4; 5; 3; 1]; [4; 3; 4; 1; 2; 5]; [5; 3; 4; 5; 3; 1]; [10; 1; 2; 3; 4; 5]]
```

```
let generation p mu tau n = tri ((tete p (mu-tau)) @ (tete (tri (enfants [] n (map tl p))) tau));;
```

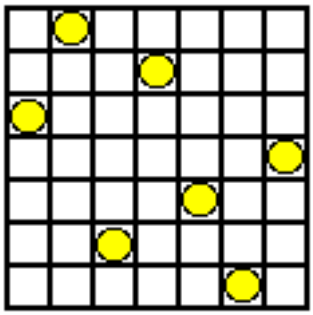
```
let rec evol gen p mu  tau  n  =  if hd (hd p) = 0 or gen = 0 then hd p
                                  else evol (gen - 1) (generation p mu tau n) mu tau n;;
```

```
let reines n gen = evol gen (init 12 n []) 12 3 n;;
```

La fonction "evol" lance l'évolution, jusqu'à un test d'arrêt, et elle sera lancée avec une population initiale au hasard. La fonction principale "reines" qui la lance, donne un résultat précédé de son évaluation 0, ce résultat résultant d'une heuristique stochastique, n'est bien entendu pas le même à chaque essai et "gen" est un nombre maximal de générations.

Ci-dessous, seconde solution [0; 3; 1; 6; 2; 5; 7; 4] en 7*7 :

```
reines 4 12;; ☞ [0; 2; 4; 1; 3]
reines 5 50;; ☞ [0; 3; 1; 4; 2; 5]
reines 5 1000;; ☞ [0; 1; 4; 2; 5; 3]
reines 6 1000;; ☞ [0; 3; 6; 2; 5; 1; 4]
reines 7 1000;; ☞ [0; 2; 7; 5; 3; 1; 6; 4]
reines 7 1000;; ☞ [0; 3; 1; 6; 2; 5; 7; 4]
reines 8 1000;; ☞ [0; 6; 3; 1; 8; 4; 2; 7; 5]
reines 9 1000;; ☞ [0; 4; 1; 7; 9; 2; 6; 8; 3; 5]
```



Remarque Lisp
On reprend les mêmes fonction avec un programme de même longueur, cependant :

```
(defun tri (q) (if (null (cdr q)) q ; tri pivot d'une liste de listes
        (sep (cdr q) (car q) nil nil)))  ; avec la tete de chaque element comme clé

(defun sep (lr p pp pg) (cond ; lr = liste restante, pp les plus petits que p, pg les plus grands
        ((null lr) (append (tri pp) (cons p (tri pg))))
        ((< (caar lr) (car p)) (sep (cdr lr) p (cons (car lr) pp) pg))
        (t (sep (cdr lr) p pp (cons (car lr) pg)))))

(defun hasard (q) (nth (random 0 (length q)) q))

(defun chrom (n r symb) (if (zerop n) r (chrom (1-  n) (cons (hasard symb) r) symb)))

(defun init (mu r ng symb) (if (zerop mu) (tri r)
                               (init (1-  mu) (cons (evaluer (chrom ng nil symb)) r) ng symb)))

(defun cross (p k m) (append (firstn k p) (nthcdr k m)))
(defun mut (c k symb) (append (firstn (1-  k) c) (cons (hasard symb) (nthcdr k c))))
```

```
(tri '((4 r a t p) (2 r a f) (6 s n c f) (3 c g t))) ☞ ((2 r a f) (3  c g t) (4 r a t p) (6 s n c f))
(chrom 10 nil '(0 1 2 3 4 5 6 7 8 9)) retournera une liste de 10 chiffres aleatoires
(chrom 5 nil '(0 1)) ☞ (0 1 0 0 1) et  (chrom 5 nil '(0 1)) ☞ (1 1 1 0 0)
Exemple sans les "scores" (init 4 nil 3 '(a t c g)) ☞ ((a t t) (c t a) (g a t) (a c c))
(cross '(a z e r t y) 3 '(u i o p q s)) ☞ (a z e p q s)
(mut '(a z e r t y) 4 '(1 2 3 4 5)) ☞ (a z e r 3 y)
```

```
(defun enfants (f p symb) ; p est la population des pères précédés de leurs scores
        (if (null p) (tri f) ; renvoie les enfants évalués et triés
        (enfants (cons (evaluer (if (zerop (random 0 2))
                (mut (cdar p) (random 0 (length (cdar p))) symb)
                (cross (cdar p) (random 0 (length (cdar p))) (cdr (hasard (if (cdr p) (cdr p) f)))))) f)
                (cdr p) symb)))

(defun generation (p mu tau symb) (tri (append (firstn (- mu tau) p) ; tau en vaeur absolue
        (firstn tau (tri (enfants nil p symb))) )))

(defun evol (gen p mu tau symb) (prin gen)(print  (car p)); petit affichage facultatif
        (if (or (zerop (caar p)) (zerop gen)) (car p) ; renvoie le meilleur tel quel, test d'arret
        (evol (1- gen) (generation p mu tau symb) mu tau symb)))

(defun run (gen ng symb) (evol gen (init 12 nil ng symb) 12 3 symb))
```

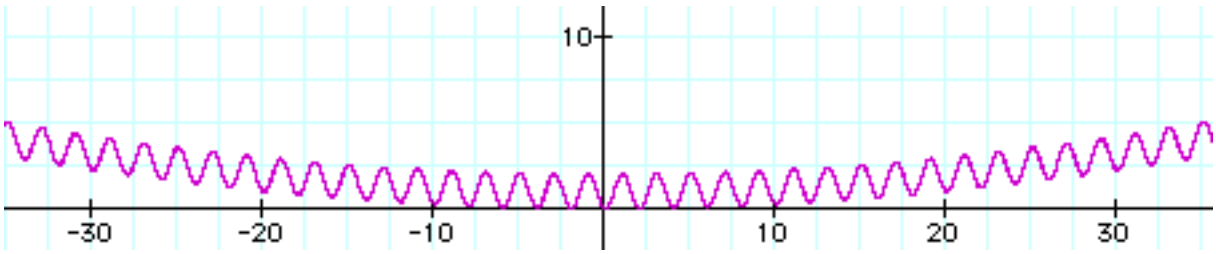
Exemple pour atteindre '(0 0 0 0 0 0 0 0)

```
(defun evaluer (c) (cons (aux c 0) c))
(defun aux (c k) (if (null c) k (aux (cdr c) (+ (car c) k))))
(de essai (gen) (run gen 8 '(0 1)))
```

Exemple de la fonction de Rastrigin

Pour une fonction de [a, b] dans R, on peut coder par la liste des décimales puis par dilatation obtenir un réel dans [a, b].
Ainsi, par exemple (7 5) représente 0,75 qui dans [1, 5] représente 4.
Cette fonction difficile à minimiser à cause de ses multiples minimums locaux est donnée en dimension 1 par :

$$F_R(x) = 0.01[x^2 + 10 - 10\cos(2\pi x)]$$




```
(defun evaluer (c) (cons (ras (+ (* 20 (decode c 0.1 0)) -10)) c)) ; ici dans [-10, 10]
(defun decode (c k r) (if (null c) r (decode (cdr c) (* 0.1 k) (+ (* k (car c)) r)) ))
(defun ras (x) (+ 2 (/ (* x x) 40) (* -2 (cos (* 6.28 x)))))
(defun essai (gen) (run gen 10 '(0 1 2 3 4 5 6 7 8 9))) ; ng = 10, reste à décoder le meilleur
```

Dames de Gauss

On reprend tout ce qui est fait en Caml :

```
(defun compte (n r) (if (zerop n) r (compte (1- n) (cons n r))))
(defun evaluer (c) (cons (gauss c 0) c))
(defun gauss (c k) (if (null c) k (gauss (cdr c) (+ (prises (car c) 1 (cdr c) 0) k) )))

(defun prises (d p c k) (cond     ; compte le nb de prises de la position d avec la liste c des col.
        ((null c) k)              ; qui sont p colonnes après elle, ex (5 7 5 2 3 4 2 1) -> 10
        ((member (car c) (list d (+ d p) (- d p))) (prises d (1+ p) (cdr c) (1+ k)))
        (t (prises d (1+ p) (cdr c) k))))

(defun reines (n gen) (run gen n (compte n nil)))
```

(reines 8 500) ☞ (0 3 5 2 8 6 4 7 1)

6. Algorithme général d'évolution avec nombreux opérateurs

Hormis les petites fonctions nécessaires au traitement du hasard et des listes, l'essentiel est dans une fonction très générale "phi" qui réalise des tirages aléatoires suivant des stratégies évolutives variées. On définit une représentation commune à tous les problèmes qui vont suivre, comme vecteur de dimension "dim" dont les composantes sont des listes de décimales. Ainsi, par simple dilatation-contraction, tout ensemble de définition $[a, b]^{dim}$ peut être ramené à $[0, 1]^{dim}$. Les contraintes liées aux différents problème seront intégrées dans la représentation.

```
let nl () = print_newline();;

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

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 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 rg x = fun [] -> 99 | (y :: q) -> (if x = y then 0 else 1 + rg x q);;
        (* = ne peut comparer des valeurs fonctionnelles *)

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);; (* q sauf ses n premiers *)

let segment p q r = tete (queue r p) (q - p + 1);; (* des rangs p à q inclus de la liste r *)

let rec compte n = if n = 0 then [] else (n - 1) :: (compte (n - 1));;

let norm x = if x < 0 then x + 9 else if x > 9 then x - 9 else x;;
        (* normalisation dans [0, 9] renvoie toujours un chiffre *)

let hasard q = nth q (random__int (list_length q)) (* renvoie un élément au hasard d'une liste *)
and has a b = a + random__int (b - a + 1)

and  has_float a b = a +. random__float (b -. a)

and 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 rec hasauf i n = let nouv = random__int n in if nouv = i then hasauf i n else nouv;;
(* donne un entier inférieur à n au hasard qui n'est pas i, donc hasauf 0 1 est impossible *)

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

let bruit x = let r = x + (if random__int 10 < 5 then 1 else (-1)) in
        if r = (-1) then 1 else if r = 10 then 8 else r;;

let rec reste p q = if p < 0 then reste (p+q) q else if p < q then p else reste (p - q) q;;
        (* reste de la division entière de p par q, le "mod" n'est pas bon *)

let maxf a b = if a<.b then b else a and minf a b = if a<.b then a else b
        and sqr x = x *. x and sqrtt x = round(sqrt(float_of_int x));; (* carré et racine entière *)

let aff t t' n = for i = 0 to n - 1 do t.(i) <- t'.(i) done and ech t i j = let z = t.(i) in t.(i) <- t.(j); t.(j) <- z;;
        (* copie, échange pour des tableaux *)

let rec puiscomp f n x = if n = 1 then f x else f (puiscomp f (n - 1) x);;
        (* composée n fois de f pour n > 0 *)
```

Nous anticipons sur le chapitre suivant, en donnant maintenant les renseignements utiles à la représentation que nous avons choisi et la visualisation des points sur un écran graphique.

```
#open "graphics";; (* sur Mac : 0 à 480 * 0 à 280 *)
let efface m = set_color white; fill_rect 0 0 m m;;
        (* petits dessins dans [0, m]2, prendre par exemple m = 100 ou 200 *)

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

let point m p a b = let x, y = (dilat m (contract p.(0) a b) a b, dilat m (contract p.(1) a b) a b )
                in set_color red; fill_rect x y 1 1;;
```

On définit, outre le type "individu", un type "opérateur" et 17 opérateurs génétiques dont celui de l'évolution différentielle (chapitre multi-agents). Tous sont des heuristiques d'exploration du domaine de définition de la fonction f que l'on veut minimiser.
Le paramètre vl désigne toujours un vecteur de listes de décimales. Ces opérateurs génétiques renvoient toujours un autre vecteur de listes. La variable locale "nc" désigne le nouveau chromosome qui est renvoyé.
Tous ces opérateurs seront ensuite partagés en unaires, binaires ou ternaires.

```
type individu = {mutable score : float; mutable genes : int list vect};;

let triv t n = (* trie-bulle d'un tableau t de n individus (à partir de 0, donc jusqu'à n-1,
        le tri étant opéré suivant le 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 t i (i + 1); drap := true) done done;;

let graph t n m a b = efface (round m); cadre (round m);
        for i = 0 to n  - 1 do point m t.(i).genes a b done;;

let bilan t n = nl(); for i = 0 to n - 1 do print_float (t.(i).score); print_string " / " done; nl();;

let vue t a b = print_string " Meilleur="; print_float t.(0).score; print_string " pour (";
        for i = 0 to vect_length t.(0).genes - 2 do print_float (contract (t.(0).genes).(i) a b);
                                                    print_string ", " done;
        print_float (contract (t.(0).genes).(vect_length t.(0).genes - 1) a b); print_char `)`;;

let migtout vl = map_vect (puiscomp (fun x -> (random__int 10) :: x) (1 + poisson 5))
                        (make_vect (vect_length vl) []);; (* renvoie un individu au hasard *)

let ajout vl = (* ajout d'une simple décimale au hasard *) let dim = vect_length vl in
        let nc = (make_vect dim []) and k = random__int dim in
        aff nc vl dim; nc.(k) <-  (nc.(k)) @ [(random__int 10)]; nc;;

let mig1 vl = (* tout est perturbé sauf les premiers éléments *)  let dim = vect_length vl in
        let nc = (make_vect dim []) in
        for i = 0 to dim - 1 do
                nc.(i) <- hd(vl.(i)) :: (puiscomp (fun x -> (random__int 10) :: x) (1 + poisson 2)) []
        done; nc;;

let mig2 vl = (* la moitié tirée au sort, des composantes est modifiée *)
        let dim = vect_length vl in let nc = (make_vect dim []) in
        for i = 0 to dim-1 do  nc.(i)<- if random__int 9 <6 then
                                        (puiscomp (fun x->(random__int 10) :: x) (1 + poisson 2))[]
                                        else vl.(i) done; nc;;
```

```
let mig3 vl = (* une composante est modifiée *)
        let dim = vect_length vl in let nc = (make_vect dim []) and k = random__int dim in
        aff nc vl dim; nc.(k) <-  (puiscomp (fun x -> (random__int 10) :: x) (1 + poisson 2)) []; nc;;

let mut1 vl = (* mutation d'un chiffre *) let dim = vect_length vl in
        let nc = (make_vect dim []) and k = random__int dim in aff nc vl dim;
        let i = random__int (list_length nc.(k)) in
        nc.(k) <-  (tete nc.(k) i) @ ((random__int 10) :: (queue nc.(k) (i+1))); nc;;

let mut2 vl = (* légère mutation d'un chiffre *) let dim = vect_length vl in
        let nc = (make_vect dim []) and k = random__int dim in
        aff nc vl dim; let i = random__int (list_length nc.(k)) in
        nc.(k) <-  (tete nc.(k) i) @ ((bruit (nth vl.(k) i)) :: (queue nc.(k) (i+1))); nc;;

let sym vl = (* 2 composantes sont échangées *) let dim = vect_length vl in
        let nc = (make_vect dim []) and k = random__int dim
        and p = random__int dim in aff nc vl dim; nc.(p) <-  vl.(k); nc.(k) <-  vl.(p); nc;;

let copie vl = (* 1 composante recopiée *) let dim = vect_length vl in
        let nc = (make_vect dim []) and k = random__int dim
        and p = random__int dim in aff nc vl dim; nc.(p) <- vl.(k); nc;;

let transpo vl = (* 2 chiffres sont échangées *) let dim = vect_length vl in
        let nc = (make_vect dim []) and k = random__int dim in
        aff nc vl dim; let d = list_length vl.(k) in if d > 3 then (let i = random__int (d - 3) in
        let j = has (i + 1) (d - 1) in
        nc.(k) <-  (tete vl.(k) i) @ [nth vl.(k) j] @ (segment (i + 1) (j - 1) vl.(k)) @ [nth vl.(k) i]
                        @ (queue vl.(k) (j + 1))) ; nc;;

let crossover0 aux vl = (* crossover uniforme *) let dim = vect_length vl in
        let nc = (make_vect dim []) in
        for i = 0 to dim - 1 do nc.(i) <- if random__int 8 < 4 then aux.(i) else vl.(i) done; nc;;

let crossover1 aux vl = (* crossover à un site *) let dim = vect_length vl in
        let nc = (make_vect dim [])  and p = random__int dim in
        for i = 0 to p do nc.(i) <- vl.(i) done; for i = p+1 to dim -1 do nc.(i) <-  aux.(i) done; nc;;

let crossover2 aux vl = (* crossover classique à 2 sites *) let dim = vect_length vl in
        let nc = (make_vect dim []) and q = random__int dim in
        let p = if q = 0 then 0 else random__int q in
        for i = 0 to p do nc.(i)<-vl.(i) done;
        for i = p + 1 to q do nc.(i)<- aux.(i) done;
        for i = q + 1 to dim-1 do nc.(i)<-vl.(i) done; nc;;

let coral = crossover2;; (* c'est un crossover avec un individu choisi parmi les 10% meilleurs *)

let reflex aux vl =  (* le symétrique de vl par rapport à aux *) let dim = vect_length vl in
        let nc = (make_vect dim []) in for i=0 to dim-1 do nc.(i)<- sym aux.(i) vl.(i) done; nc
        where rec sym = fun _ [] -> [] | [] _ -> [] | (x :: q) (y :: r) -> (norm (2*x - y)) :: (sym q r);;

let de aux aux' vl = (* opérateur différentiel, sorte de tri-crossover *) let dim = vect_length vl in
        let nc = (make_vect dim []) in for i = 0 to dim - 1 do nc.(i)<- h vl.(i) aux.(i) aux'.(i) done; nc
        where rec h = fun [] _ _ -> [] | _ [] _ -> [] | _ _ [] -> []
                        | (y :: q) (z :: r) (x :: s) -> (norm (x + (y - z) / 2)) :: (h q r s);;

let decoral best aux p = de p aux best;; (* x -> x + khi(y - z) où y, z autres parents et khi =1 / 2 *)
```

L'opérateur "coral" provient de l'observation de certains poissons de corail qui ne se reproduisent qu'avec une minorité (10%) de leurs congénères. "reflex" est une sorte de symétrie centrale, "de" provient de l'heuristique dite d'évolution "différentielle", et enfin "decoral" est un opérateur différentiel qui ne se produira qu'avec un partenaire parmi les 10% meilleurs.

Vient ensuite une définition de la proximité entre vecteurs qui permettra un certain éclaircissement dans la population : La fonction "elim" balaie un tableau trié t à l'envers, des plus grandes valeurs de f vers les plus faibles. Dès que deux individus du tableau sont proches à moins de "prox", (qui sera de l'ordre de 33%), le moins bon est remplacé grâce à l'opérateur "migtout" par un individu entièrement aléatoire.
Cette méthode très simple permet de résoudre le dilemme entre l'exploitation des bonnes solutions déjà obtenues, et l'exploration de "l'espace de recherche" où pourraient se trouver de meilleures solutions pour la minimisation de f.

```
let proches v v' = (* en %, proches' 0 0 [1;2;3;4;5;6;7;8;9;0] [1;2;3;4;8;5;6;7;8;9;0] = 40 *)
        let rec proches' nb lg = fun [] _ -> (100 * nb) / lg | _ [] ->(100 * nb) / lg
                        | (x :: q) (y :: q') ->proches' (nb + (if x = y then 1 else 0)) (lg + 1) q q'
        in let m = ref (proches' 0 0 v.(0) v'.(0)) in
        for i = 1 to vect_length v - 1 do m := min (!m) (proches' 0 0 v.(i) v'.(i)) done; !m;;
let elim t f a b n prox = let k = ref 0 in for i = n - 1 downto 1 do
        if prox < proches t.(i).genes t.(i - 1).genes
        then (incr k; let g = migtout t.(i).genes in
                        t.(i) <-  {score = f (map_vect (fun x -> contract x a b) g); genes = g}) done;
        triv t n; !k;;
(* A cause de "triv ", "elim" retrie toujours t et renvoie le nb d'évaluations supplémentaires de f *)
```

A présent, on définit le type "opérateur" avec les fonctions associées à ce type. "Nom" renvoie son nom, "valop" renvoie son score, car ces opérateurs vont être eux-mêmes évalués, "init" permet de leur attribuer un score initial (qui sera 0) et "augm" est une fonction qui pourra modifier ce score.

```
type Operateur = Op1 of float*string*(int list vect -> int list vect)
        | Op2 of float*string*(int list vect -> int list vect -> int list vect )
        | Op3 of float*string*(int list vect -> int list vect -> int list vect -> int list vect );;

let nom = function Op1(_, n, _) -> n | Op2(_, n, _) -> n | Op3(_, n, _) -> n
and augm = function Op1(x, n, f) -> Op1(x+.1., n, f)
                        | Op2(x, n, f) -> Op2(x +. 1., n, f)
                        | Op3(x, n, f) -> Op3(x +. 1., n, f)
and init b = function Op1(_, n, f) -> Op1(b, n, f)
                        | Op2(_, n, f) -> Op2(b, n, f)
                        | Op3(_, n, f) -> Op3(b, n, f)
and valop = function Op1(v, _, _) -> v | Op2(v, _, _) -> v | Op3(v, _, _) -> v ;;

let trip t n = (* tri-bulle d'un tableau t de n opérateurs, c'est une stupide contrainte du typage *)
        let drap = ref true and k = ref (n - 1) in
        while !drap do drap := false; decr k; for i = 0 to !k do
                if valop t.(i + 1) <. valop t.(i) then (ech t i (i + 1); drap := true) done done;;
```

On définit un certain nombre de variables globales, comme le tableau "operef" des opérateurs de référence, et le tableau "oper" qui va être modifié suivant les méthodes (le paramètre m du 9) employées.
Enfin, pour réunir ces différentes méthodes, la fonction "applic" applique effectivement l'opérateur de numéro "indop" sur l'individu de rang i dans le tableau tp, de taille n, avec la méthode m. Afin de comparer les scores et de renvoyer un booléen indiquant s'il y a amélioration, cette fonction utilise les paramètres f (la fonction à minimiser) et l'intervalle [a, b].

```
let operef = [| Op1(0., "mut1", mut1); Op1(0., "mut2", mut2);
        Op1(0., "ajout", ajout);
        Op1(0., "mig1", mig1); Op1(0., "mig2", mig2); Op1(0., "mig3", mig3);
        Op1(0., "sym", sym); Op1(0., "transpo", transpo); Op1(0., "copie", copie);
        Op1(0., "migtout", migtout);
        Op2(0., "crossover0", crossover0); Op2(0., "crossover1", crossover1);
        Op2(0., "crossover2", crossover2); Op2(0., "reflexion", reflex);
        Op2(0., "coral", coral); Op3(0., "de", de); Op3(0., "decoral", decoral) |];;
```

```
let no = vect_length operef;; (* constante nombre d'opérateurs, il y en a 10 qui sont unaires *)

let oper = make_vect no (Op1(0., "mut1", mut1))
and oprob = make_vect no (0., 0.);;  (* vecteur servant aux probabilités des opérateurs *)

let applic tp i n f a b m indop = let g = (match oper.(indop) with
        | Op1(s, _, op) -> op
        | Op2(s, nom, op) -> op tp.(if nom = "coral" then random__int (1 + n /10)
                                                    else hasauf i n).genes
        | Op3(s, nom, op) -> op tp.(if nom = "decoral" then random__int (1 + n /10)
                                                    else hasauf i n).genes tp.(hasauf i n).genes)
        (tp.(i).genes) in (* calcul du fils g de tp.(i) , g = nouveaux genes, calcul de sa  fitness : *)
        tp.(i) <- {score = f (map_vect (fun x -> contract x a b) g); genes = g};
        let s' = tp.(i).score-.tp.(n+i).score in
        if m>3 then (oper.(indop) <-  (match oper.(indop) with Op1(s, n, op) -> Op1(s+.s', n, op)
                        | Op2(s, n, op)->Op2(s+.s', n, op) | Op3(s, n, op) -> Op3(s+.s', n, op)));
        if m=0 then (let (k, d) = oprob.(indop) in oprob.(indop) <-  (k +. 1., d +. s'));
        s' <. 0. ;;

(* "applic" modifie toujours le score de l'opérateur et renvoie un booléen "amélioration", n est la
taille de la population, i l'indice de l'individu dans la table tp (de taille n) , sur lequel s'applique
l'opérateur d'indice "indop", m est la méthode définie en 9 *)

let modif mieux = (* empile dans le vecteur "oper"  *)
        if mieux then (for i = no - 1 downto 1 do oper.(i) <- oper.(i-1) done; augmente oper.(0))
        else (for i = 0 to no - 1 do oper.(i) <- init 0. operef.(i) done) (* ou bien le réinitialise *)
        where augmente x = for k = 0 to no - 1 do
                        if nom operef.(k) = nom x then operef.(k) <- augm operef.(k) done ;;

let compete () = (* donne un numero d'opérateur suivant sa proba, calcul avec proba cumulées *)
        let k = ref 0 and fin = ref false and p = ref 0. in
        while not (!fin) & !k < no do
                p := !p +.  (valop oper.(!k));
                if random__float 1. < !p then fin := true else incr k done; !k;;
```

7. L'algorithme génétique standard

L'algorithme génétique classique est caractérisé par une volonté de brouiller les individus par un codage binaire. Une mutation peut alors donner un individu aussi bien voisin qu'éloigné. Ici la fonction "ga" s'applique au contraire à la repésentation définie en 6. La démarche la plus courante consiste à croiser un individu (suivant une probabilité de l'ordre de $p_C = 0{,}6$), exercer une mutation (avec une probabilité de l'ordre de $p_m = 0{,}1$) sur les deux enfants et remplacer les parents par les enfants. Cette heuristique qui fut la première, s'inspire de la génétique darwinienne des espèces vivantes où, statistiquement, les individus les plus aptes, suivant un critère à définir, ont la meilleure longévité, et donc le plus de chance de se reproduirent et de transmettre leur particularité génomique, laquelle va donc se propager au fil des générations. Cependant, pour reproduire assez fidèlement, quoiqu'en la simplifiant énormément, l'évolution naturelle, cette heuristique risque de perdre ses meilleurs individus (elle n'est pas élitiste) et par conséquent d'être cantonée dans un minimum local de f.

Voir Goldberg D.E. *Genetic algorithms in search, optimization and machine learning,* Addison Wesley, 1989

La fonction suivante donne le nombre d'évaluations pour obtenir une solution x dans $[a, b]^{dim}$ telle que f(x) < eps. On applique l'opérateur numéro 12 (crossover) puis avec la probabilité p_m l'opérateur 1 (mutation). L'argument m = 1 est un numéro de méthode d'application des opérateurs dont le détail figure dans la fonction "phi" plus loin. Le paramètre "max" est le nombre maximal d'évaluations qu'on s'autorise et "clr" est le taux en dessus duquel on pratique une élimination.

```
let ga mu max eps f a b dim pc pm clr visu =
let ng = ref 0 and nv = ref 0
and tp = make_vect (2 * mu)  {score = 0.; genes = make_vect dim []} in
        (* ng = nombre de génération, nv = nombre d'évaluations de f, tp = tableau - population *)
for i = 0 to mu - 1 do
        let g = migtout tp.(0).genes in (* population initiale de mu chromosomes *)
        tp.(i) <- {score = f (map_vect (fun x ->contract x a b) g); genes = g}
        done;
        nv := !nv + mu; triv tp mu;
        if visu then (print_string "ng = 0"; vue tp a b);
while !nv < max & eps <. tp.(0).score (* conditions d'arrêt *)do
        for i = 0 to mu - 1 do let drap = ref false in
        if random__int mu < mu - i (* sélection par le rang *) & random__float 1. < pc
        then (drap := applic tp i mu f a b 1 12; incr nv;
                if random__float 1. < pm then drap := applic tp i mu f a b 1 0; incr nv) done;
        triv tp mu; incr ng; nv := !nv + elim tp f a b mu clr; (* "elim" tue des individus, les remplace
                par de nouveaux, et donc augmente le nombre d'évaluations de f *)
        if visu then (nl(); print_string ("ng = " ^ (string_of_int !ng) ^ " nv = "^ (string_of_int !nv));
                        vue tp a b )
done; !nv;;
```

8. Les stratégies d'évolution de Schwefel

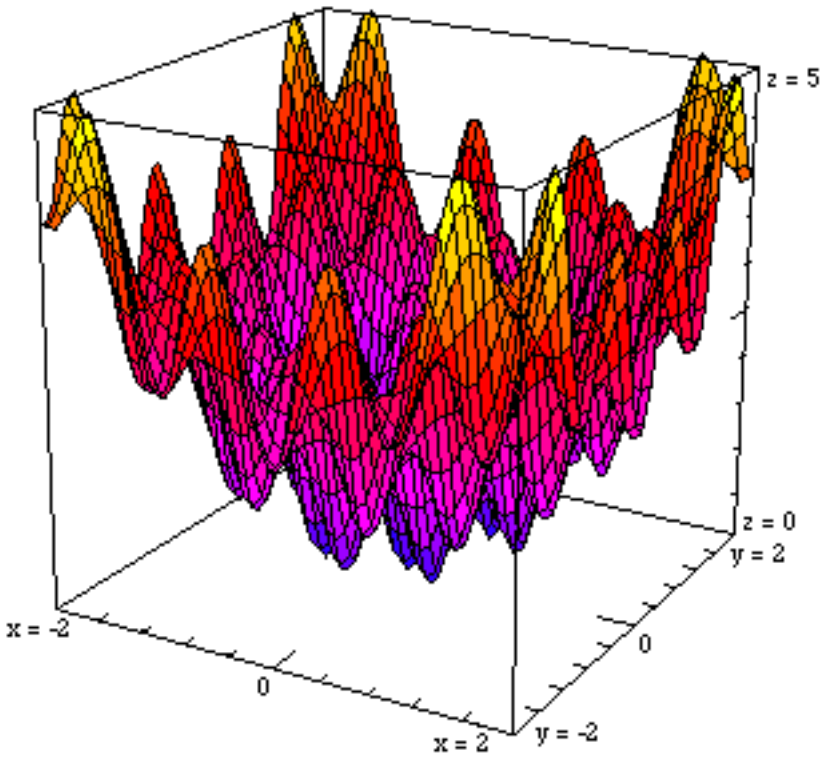
Les stratégies d'évolutions consiste avec une représentation réelle (ici, toujours la même) et une population faible de taille μ, à produire à chaque génération, λ enfants par parents (le paramètre lb). A l'issue d'une génération, la stratégie dite élitiste construit la nouvelle population en retenant les μ meilleurs de la population rassemblée parents + enfants. La stratégie dite non élitiste, elle, prend les μ meilleurs uniquement dans les $\mu\lambda$ enfants.
Cette heuristique, s'éloigne de la nature en favorisant une sorte d'accélération de l'évolution. C'est pourquoi, elle a généralement de meilleurs résultats que GA, surtout la stratégie élitiste nommée ES(μ + λ).

Voir : Bäck T. Fogel D.B. Schwefel H.P. *Handbook of evolutionnary computation,* Oxford University Press, 1997

```
let es mu max eps f a b dim lb op elit clr visu = (* elit = stratégie élitiste *)
        let ng = ref 0 (* ng = nb de générations, nv = nb d'évaluations de f *)
        and nv = ref 0 (* utilisation de tp pour les mu parents + mu * lbd enfants *)
        and tp = make_vect ((lb + 1) * mu)  {score = 0.; genes = make_vect dim []}
        in (* tp = tableau - population de taille prévue pour parents et progéniture *)
        for i = 0 to mu - 1 do let g = migtout tp.(0).genes (* initialisation *)
                        in tp.(i) <- {score = f (map_vect (fun x ->contract x a b) g); genes = g}
        done;
        nv := !nv + mu; triv tp mu; if visu then (print_string "ng = 0"; vue tp a b);
while !nv < max & eps <. tp.(0).score do let drap = ref false in (*  parents recopiés lb fois *)
        for i = 0 to mu - 1 do for k = 1 to lb do tp.(k * mu + i) <- tp.(i) done done;
        for i = 0 to mu-1 do for k = 0 to lb - 1 do
        drap := applic tp (k*mu+i) mu f a b 1 (if op then random__int no (* stratégie "op" *)
                                else (if random__float 1. < 0.5 then 0 else 12)) (* mut + cross *)
        done done;
        nv := !nv + mu * lb; (* la fonction renvoie encore le nombre d'évaluations de f *)
        incr ng;
        if elit then triv tp (if elit then (lb + 1)*mu else lb*mu); nv := !nv + elim tp f a b mu clr;
        if visu then (nl(); print_string ("ng =" ^ (string_of_int !ng) ^ " nv = " ^ (string_of_int !nv));
                        vue tp a b )
done; !nv;;
```

Les fonctions sur lesquelles on teste des comparaisons entre "ga", "es" sont d'abord des fonctions mathématiques dont on connaît d'avance le minimum 0.

Ainsi, la fonction de Griewank ci-dessous représentée en dimension 2.



$$F_G(x) = (1/4000)\sum_{1 \le i \le dim} x_i^2 - \prod_{1 \le i \le dim} \cos(x_i / \sqrt{i}) + 1, \text{ ici sur } [-2, 2]^2$$

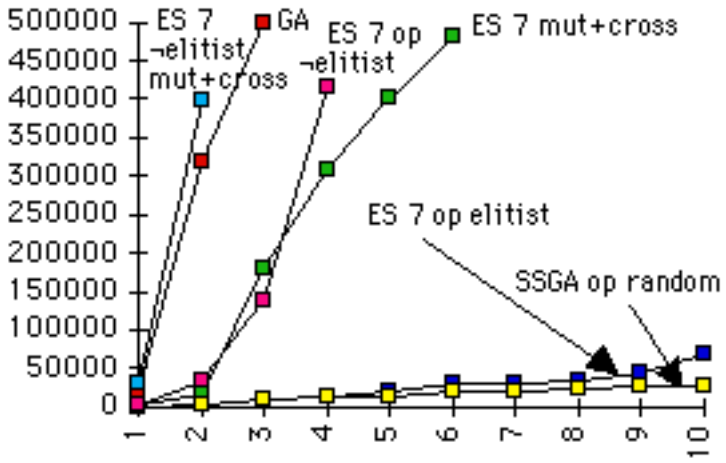
```
let gri x = let s = ref 0. and p = ref 1. in for i = 0 to vect_length x  - 1 do
        s := !s +. sqr(x.(i)); p := !p *. (cos ((x.(i)) /. sqrt (1. +. float_of_int (i)))) done;
        1. +. !s /. 4000. -. !p;;

let parab x = let r = ref 0. in for i = 0 to vect_length x  - 1 do r := !r +. sqr(x.(i)) done; !r;;
        (* Fonction paraboloïde = Σ(xi)^2 *)

let jong x = let s = ref 0 in for i = 0 to vect_length x - 1 do
        s := !s + (abs (round x.(i))) done; float_of_int !s;;
        (* Fonction de De Jong FJ(x) = Σ |round(xi)| si x ∈ [-50, 50]^n *)

let ros x = 100. *. sqr(sqr(x.(0)) -. x.(1)) +. sqr(1. -. x.(0));;
        (* Fonction définie en dimension 2 de Rosenbrock 100(x^2 - y)^2 + (1 - x)^2 sur [-2, 2]^2 *)

let ras x = let s = ref 0. and p = ref 1. in for i = 0 to vect_length x -1 do
        s := !s+.sqr(x.(i)) +. 10. -. 10.*.(cos (6.28*.x.(i))) done;!s;;
        (* Fonction de Rastrigin déjà évoquée en 5 *)
```

On lance par exemple :

```
es 10 1000 1e-04 gri (-500.) 500. 1 7 true true 33 true;;
ga 10 1000 1e-04 gri (-500.) 500. 2 0.5 0.1 33 true;;
```

Les comparaisons sont effectuées en prenant mutation + crossover, ou bien l'ensemble des opérateurs (courbes "op"). On prend $\lambda = 7$ qui est expérimentalement optimal, et avec l'algorithme SSGA nanti des 17 opérateurs appliqués au hasard, (problème 9).
On demandera, pour la fonction de Griewank, d'allure semblable à Rastrigin vue en 5, mais plus difficile, de lancer 20 fois "ga" et "es".

Les graphiques suivants montrent les moyennes de ces nombre d'évaluations pour atteindre 10^{-4} dans le domaine $[-500, 500]^{dim}$ pour la fonction de Griewank à des dimensions $1 \leq dim \leq 10$.



Comme on le voit, la multiplication des opérateurs comme méthodes d'exploration de l'espace de définition de la fonction à minimiser est toujours préférable à l'emploi des simples croisement et mutation. Par ailleurs, l'élitisme est également toujours préférable.

9. La stratégie régulière SSGA avec ou sans tri et compétition d'opérateurs

La stratégie SSGA définie en 5, est brièvement rappelée ici. Partant d'une population aléatoire de taille mu, ses individus représentent des solutions éventuelles minimisant la fonction f.
Le paramètre taux est la mesure du renouvellement (0 variable, 100 on perd l'élitisme, et par exemple 33 peut être recommandé).
A chaque génération, les μ parents produisent μ enfants en utilisant ou non tous les opérateurs définis en 6, et le taux détermine la part des parents les moins bons qui vont être remplacés par les meilleurs enfants pour constituer la génération suivante.
Le paramètre m indique une méthode d'application des opérateurs, ses valeurs sont :

0 : les opérateurs recoivent une probabilité d'application, mise à jour à chaque génération, (méthode "compete" qui utilise la fonction "compete").

1 : uniquement les opérateurs unaires
Cette méthode donne souvent de bons résultats dans des problèmes symboliques où les croisements peuvent faire perdre du temps à la recherche de bonnes solutions.

2 : opérateurs appliqués suivant le rang où ils sont rangés dans "oper" (méthode "rank").

3 : application au hasard ("random")
C'est le plus souvent, la méthode la plus robuste : faire confiance au hasard.

4 : les opérateurs sont triés suivant leur performance cumulée, à faire décroitre f et sont d'autant mieux appliqués qu'ils sont parmi les premiers (roulette biaisée, "wheel").

5 : opérateurs simplement triés suivant leur performance (méthode "sort").

6 : le tri est assorti d'une duplication du meilleur opérateur au début de la famille des opérateurs, sauf au cas où aucune amélioration pour f est observé, alors les opérateurs de référence sont réinitialisés, (méthode "duplication" utilisant la fonction "modif").

Venons-en maintenant à la fonction principale phi. Sans doute, la plus longue du livre, mais il faut dire qu'elle mêle plusieurs heuristiques. Cette fonction renvoie le nombre d'évaluations (inférieur à max) de f définie sur $[a, b]^{dim}$ pour obtenir une solution entre 0 et eps.

Les paramètres mu et taux sont respectivement la taille de la population et le taux de renouvellement exprimé en pourcentage. La variable locale "tx" est ce taux en absolu. La variable "amp" détermine l'amplitude de f afin de mettre en oeuvre des taux de renouvellement variables, mais cette idée n'est pas concluante.
Le paramètre clr est le taux d'élimination, dès que proximité > clr, entre un individu et son suivant dans le tableau tp des individus, alors on remplace le moins bon d'entre eux par un individu totalement aléatoire (clr = 100 pas d'élimination, et clr = 0 détermine un taux variable).

```
let phi mu max eps f a b dim taux clr m visu =
        (* mu=taille popu > 10, max, eps cond. de stop, f définie sur [a, b]^dim *)
        if visu then (open_graph" "; while not key_pressed () do print_int (random__int 9) done;
                nl()); (* un moyen simple de faire partir la série des nombres aléatoires *)
        let ng = ref 0 and nv = ref 0
        and tp = make_vect (2*mu) {score = 0.; genes = make_vect dim []} in aff oper operef no;
        for i = 0 to mu-1 do let g = migtout tp.(0).genes in (* initialisation de la population *)
                tp.(i) <- {score = f (map_vect (fun x -> contract x a b) g); genes = g} done;
        nv := !nv + mu; if visu then (print_string "ng=0"; graph tp mu 100. a b); triv tp mu;
        if visu then vue tp a b;
if m = 0 then for i = 0 to no - 1 do oper.(i) <- init (1. /. (float_of_int no)) operef.(i) done;
let amp, tx = tp.(mu-1).score -. tp.(0).score, ref (mu * taux / 100) in
while !nv < max & eps <. tp.(0).score do let drap = ref false and k = ref 0 in
        for i = 0 to mu - 1 do tp.(mu+i) <- tp.(i); (*  parents entre μ et 2μ, enfants entre 0 et μ *)
                drap := !drap or applic tp i mu f a b m (match m with
                | 0 -> compete () (* competition*)
                | 1-> has 0 9 (* les 10 op. unaires *)
                | 2 -> incr k; (!k mod no) (* rang *)
                | 3 -> random__int no (* hasard *)
                | 4 -> int_of_float ((sqr(random__float 1.))*.(float_of_int no)) (* roulette *)
                | _ -> i mod no (* tri et duplication *)) done;
        if m = 0 then (let v  = ref 0. and s = ref 0. in (* cas de la statégie "compete" *)
                for i = 0 to no - 1 do
                let k, d = oprob.(i) in (let w = if k = 0. then 0. else d /. k in oprob.(i) <-  (0., w);
                        if !v < w then v := w; s := !s +. w) done;
                v := !v  +. 0.01;
                for i = 0 to no - 1 do oper.(i) <-
                        init (((!v) -. (snd oprob.(i))) /. ((float_of_int no) *. (!v) -. !s)) operef.(i) done);
                (* chq op. ds le cas m=0 se voit calculé son amélioration en moyenne dans [u, v]
                lors de la génération écoulée, puis normalisation dans oper *)
        if m > 3 then (trip oper no; if m > 5 then modif !drap); (* modification des opérateurs *)
        nv := !nv + mu; incr ng; triv tp mu;
        for i = !tx to mu - 1 do tp.(i) <- tp.(mu + i -  !tx) done; triv tp mu;
                (* on garde tx meilleurs enfants, mu-tx meilleurs parents et on retrie le tout *)
        if taux = 0 then (* cas où fait un taux de renouvellemnt variable, non pertinent *)
                tx := round (10. +.80.*.(minf 1. (tp.(mu -.1).score -. tp.(0).score) /.amp))* mu / 100;
        if clr < 100 then nv := !nv + elim tp f a b mu (if clr = 0 then !tx else clr);
                (* clr = 0 : élimination variable : non pertinent, elim retrie toujours après *)
        if visu then (nl(); print_string ("ng = " ^ (string_of_int !ng) ^ " nv = " ^ (string_of_int !nv)
                                ^ " moyenne = " );
                let s = ref 0. in for i = 0 to mu - 1 do s := !s +. tp.(i).score done;
                print_float (!s /. (float_of_int mu)); vue tp a b; graph tp mu 100. a b; bilan tp mu;
                for i = 0 to no - 1 do print_string (nom oper.(i)); print_string " ";
                                print_float (valop oper.(i)); print_string ", " done; nl())
done; if visu then (nl(); close_graph()); !nv;;
```

Exemple	phi 8 1000 1e-06 jong (-500.) 500. 2 50 33 6 true;;

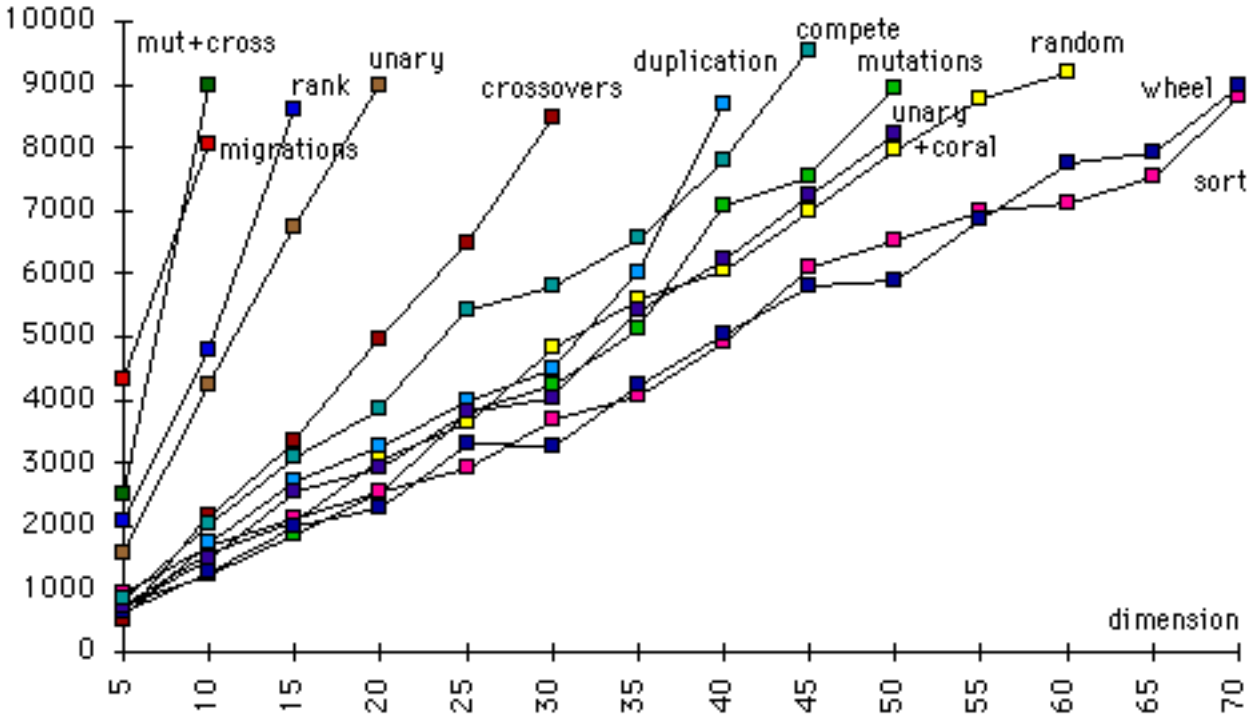
10. Application à la comparaison de méthodes pour optimiser la fonction de Jong

C'est la fonction simple, mais en paliers, définie par : Jong(x) = $\sum$ |round(x_i)| si x $\in$ $[-50, 50]^{dim}$. Son intérêt réside, comme pour le problème de la "voie royale" évoqué plus loin, en ce que de petites modifications de x peuvent ne rien changer ou bien apporter un grand changement.

L'optimisation de cette fonction en dimensions 5 à 70 se fait suivant une longue expérimentation avec la fonction "averag" donnant la moyenne de n tirages de "phi" appliquée à "jong".

Nous choisissons ici $\mu = 9$ et $\tau = 3$, et grâce à "averag", et lançons une très longue recherche.

```
let jong x = let s = ref 0 in
        for i = 0 to vect_length x -  1 do
        s := !s + (abs (round x.(i))) done;
        float_of_int !s;;

let averag mu max eps f a b dim taux clr m n = aff oper operef no; let s = ref 0 in
        for i =1 to n do s := !s + phi mu max eps f a b dim taux clr m false done;
        !s / n;;

    for m = 0 to 6 do nl(); print_int m; print_string " -> ";
    for n = 1 to 14 do nl(); print_string " dim = "; print_int (5 * n); print_string " ";
    print_int (averag 9 200000 1e-06 jong (-50.) 50. (5 * n) 35 33 m 20) done done;;
    (* résultats présentés graphiquement *)
```



Ce graphique montre la moyenne du nombre d'évaluations de la fonction "jong" pour arriver à son minimum 0, suivant 12 méthodes : les 7 indiquées à choisir par le paramètre m de phi, plus quelques-unes qu'il est facile de redéfinir (les crossovers uniquement, les migrations uniquement et mutation + crossover).

Le fait de prendre les opérateurs au hasard (courbe "random") est de manière générale presque la meilleure méthode (faire confiance au hasard). Mais elle est légèrement dépassée ici par les méthodes consistant à trier les opérateurs suivant leur performance. Avec ces méthodes les meilleurs opérateurs sont appliqués aux meilleurs individus.

11. Dames de Gauss n sur n.

Il s'agit toujours de placer n dames sur un damier de n sur n, sans qu'il y ait de prises possibles, en cherchant à minimiser le nombre de paires de dames en position de prise, pour l'amener à 0. Nous choisissons une façon de représenter un damier n*n de la façon suivante :

Dans $[0,1]^n$, le numéro de la ligne dans la colonne i est tout simplement $E(n.x_i)$, où E est la partie entière, ce que fait la fonction "ligne". On choisit, toujours avec le même "phi" et la même fonction "averag", ici les paramètres sont $\mu = 8$ et $\tau = 4$, et pour comparer les méthodes, la moyenne est faite sur 50 runs.

```
let lignes x = let dim = vect_length x in map_vect (fun x -> int_of_float(x *. (float_of_int dim))) x ;;


let dames x = let dim = vect_length x and lgn = lignes x and s = ref 0 in
        for i = 0 to dim - 1 do for j = i + 1 to dim - 1 do
                if lgn.(i) = lgn.(j) or abs (i - j) = abs (lgn.(i) - lgn.(j)) then incr s done done;
        (float_of_int !s);;
```

```
lignes [|0.1; 0.0; 0.45; 0.85; 0.23; 0.587; 0.12; 0.96; 0.4; 0.02; 0.8|]
☞ [|1; 0; 4; 9; 2; 6; 1; 10; 4; 0; 8|]

for n = 5 to 30 do print_int n; print_string " -> ";
print_int (averag 8 500000 1e-06 dames 0. 1. n 50 33 4 50); nl() done;;
(* résultats présentés graphiquement en variant m *)
```

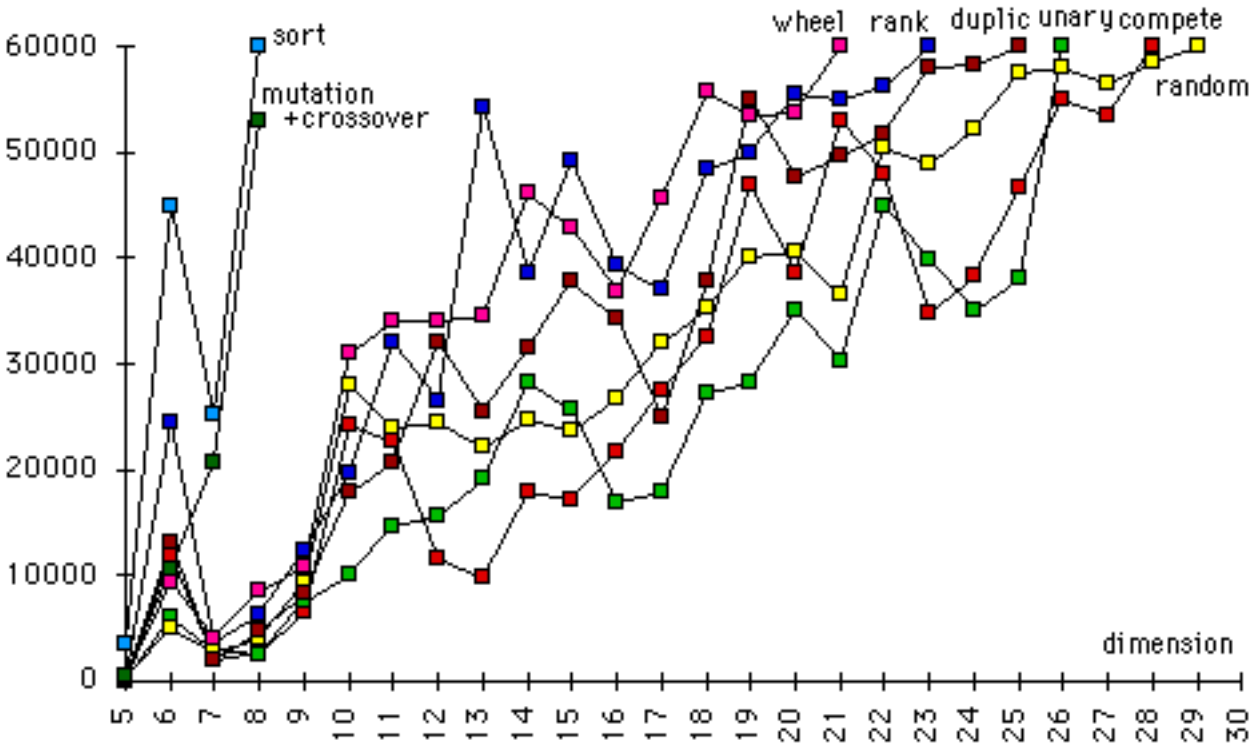
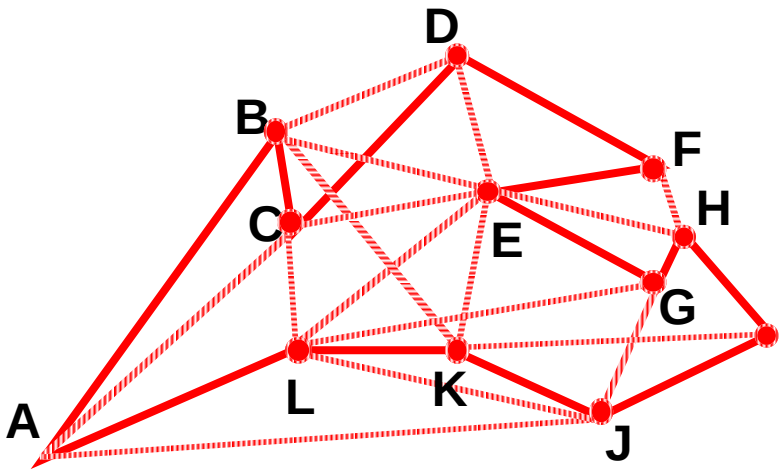


Illustration de la moyenne du nombre d'évaluations de la fonction "dames" pour arriver à son minimum 0, suivant ssga(8, 4) et les 7 méthodes indiquées à choisir par le paramètre m de phi, plus "mutation + crossover". Les damiers vont de 5*5 à 30*30.

Le fait de prendre les opérateurs au hasard (courbe "random") semble assez fiable, quoique le fait de ne pas prendre de crossovers (méthode "unary") ne soit pas mauvais.

12. Voyageur de commerce sur n villes

Le problème classique du voyageur de commerce est simple à définir. Soient n villes x_i à parcourir en trouvant la boucle (avec retour au départ) la plus courte ne passant pas deux fois par la même ville et n'en manquant aucune. (On reverra plus loin ce problème résolu par colonie de fourmis ou par la méthode de l'élastique).
Ci-dessous une disposition de 12 villes.

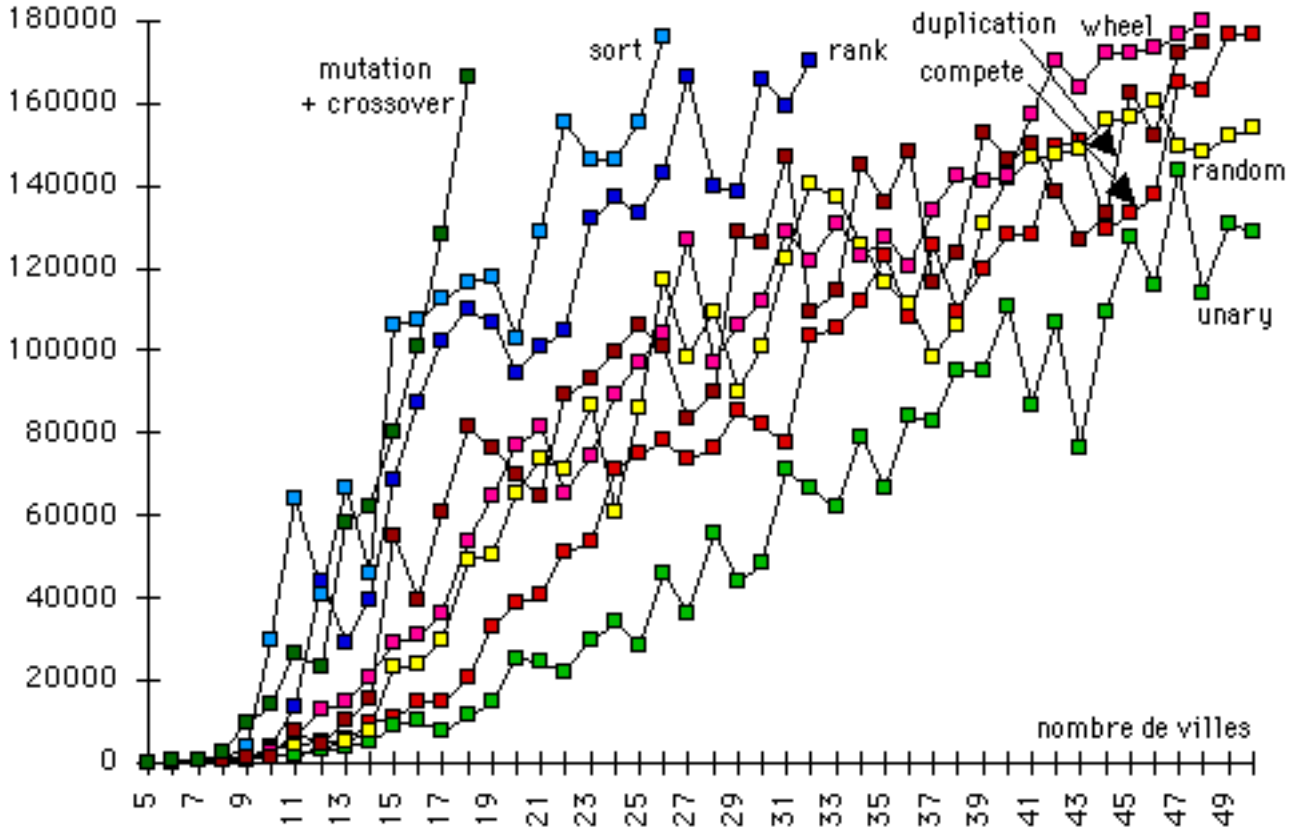


Là encore, la difficulté de revoir des opérateurs adaptés au problème nous pousse à convenir d'une représentation cadrant avec celle qui précède.

Pour $x \in [0,1[^n$, la 1° ville est $E(n.x_1)$, et :
la ville numéro i est le $E((n - i + 1)x_i)$-ième élément de [1..n] - $\{v_1 ... v_{i-1}\}$ avec un chemin optimal connu, revient à trouver une permutation particulière parmi les n! permutations.
Grâce à cette représentation, les contraintes de passer une fois et une seule fois en chaque ville, sont satisfaites.

Pour l'exemple ci-dessous on a dim = 7 et dans le tableau des villes parcourues, on obtient la valeur du paramètre s = 5+2+1+3+1+3 = 15 d'où tsp = 15 - 7 + 1 = 9.
On applique le ssga(12, 4) à la fonction "tsp".

```
let villes x = let n = vect_length x in
        let v = make_vect n 0 and ind = ref (compte n) in
        for i = 0 to n-1 do v.(i) <- nth !ind (int_of_float(x.(i)*.(float_of_int (n - i))));
                                ind := ret v.(i) !ind
        done;
        v;;

villes [|0.2; 0.899; 0.7523; 0.785; 0.63; 0.5321; 0.047 |];;
☞  [|5; 0; 2; 1; 4; 3; 6|]

let tsp x = let dim = vect_length x and v = villes x and s = ref 0 in
        for i = 0 to dim - 2 do s := !s + abs(v.(i) - v.(i + 1)) done;
        float_of_int (!s - dim + 1);; (* consiste à faire un tour dans l'ordre pour avoir 0 *)

tsp [|0.2; 0.899; 0.7523; 0.785; 0.63; 0.5321; 0.047 |];;
☞  9.0

Pour comparer : for m = 0 to 6 do nl(); print_int m; print_string " -> ";
for n = 1 to 50 do nl(); print_string " dim = "; print_int n; print_string " ";
print_int (averag 12 200000 1e-06 tsp 0. 0.99 n 35 33 m 20) done done;;
(* graphique ci-contre *)
```



Moyenne du nombre d'évaluations de la fonction "tsp" pour arriver à son minimum, suivant ssga(12, 4) et les mêmes méthodes d'application des opérateurs.
Le fait de prendre les opérateurs au hasard (courbe "random") est encore assez fiable, quoique dépassé par la méthode "unary" (sans les crossovers).

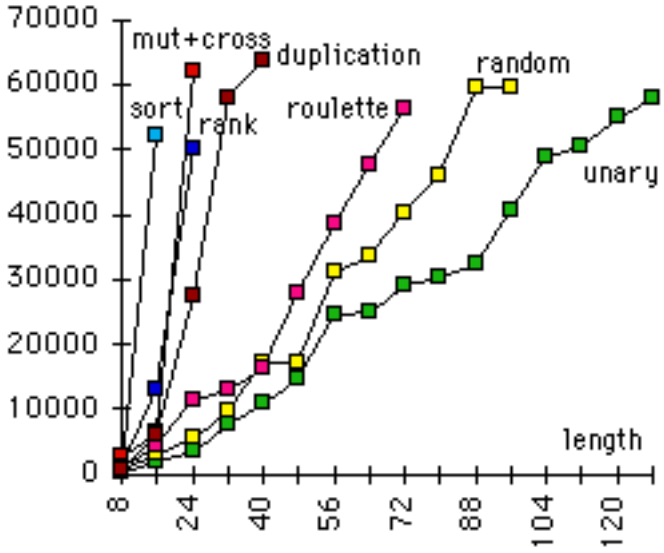
13. Problème de la voie royale

Il s'agit de compter les sous-chaînes de 8 bits uniformes à 1. Ce problème a été construit pour tester des algorithmes génétiques, il est difficile dans la mesure où une petite mutation de 0 en 1 modifie tout un plateau et fait sauter la fonction à optimiser vers une valeur discrète bien différente.
Pour $x \in [0,1]^n$, (en fait dans {0,1} avec round), chaque sous-chaine consécutive de 8 bits à 1 aux positions multiples de 8, compte pour 8, puis on fait une différence de façon à se ramener encore à un problème de minimisation dont on connait le minimum 0. Ici, n est multiple de 8.

Voir : Mitchell M. Forrest S. Holland J.H. *The royal road for genetic algorithm : fitness landscapes and GA performances,* Varela F.J. Bourgine P. Ed. 1992

```
let rr x = let n = vect_length x and c = ref 0 in for i = 0 to n / 8 -1 do
        if sum (sub_vect x (8 * i) 8) = 8 then c := !c - 8 done;
        float_of_int (n + !c)
where sum v = let s = ref 0 in for i = 0 to 7 do s := !s + round v.(i) done; !s;;
```

```
rr [|0.4; 0.2; 0.1; 0.3; 0.4; 0.2; 0.1; 0.3; 0.4; 0.2; 0.1; 0.3; 0.4; 0.2; 0.1; 0.3; 0.4; 0.2; 0.1;
0.3; 0.4; 0.2; 0.1; 0.3; 0.7; 0.8; 0.9; 0.6; 0.6; 0.7; 0.7; 0.8; 0.7; 0.8; 0.9; 0.6; 0.6; 0.7; 0.7;
0.8|];; ☞ 24.0
for m = 0 to 6 do nl(); print_int m; print_string " -> ";
for n = 1 to 8 do nl(); print_string " dim = ";
print_int (8 * n); print_string " "; print_int (averag 12 200000 1. rr 0. 1. (8 * n) 35 33 m 10)
done done;; (* graphique ci-après *)
```



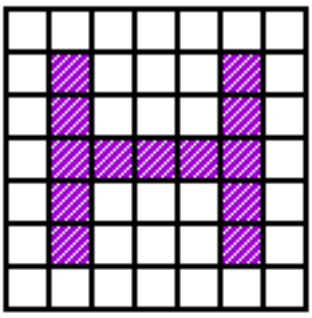
Moyenne du nombre d'évaluations de la fonction "rr" pour arriver à son minimum 0, suivant l'algorithme ssga (8, 4). Le fait de prendre les opérateurs au hasard (courbe "random") est encore dépassé par le fait de ne pas prendre de crossovers (méthode "unary").

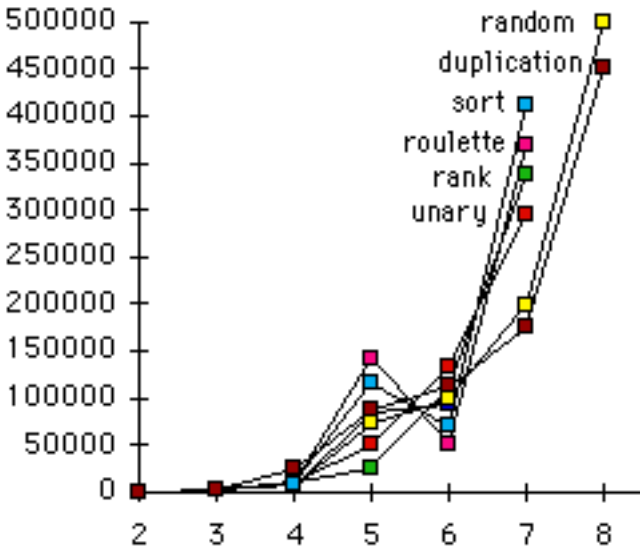
14. Reconnaissance d'une figure comme Θ en n sur n.

Il s'agit simplement de reconnaître sur des tableaux binaires carrés de n sur n, une figure représentant le dessin d'une lettre par exemple. La représentation d'une solution est toujours un vecteur de liste de chiffres, lesquelles listes sont, cette fois, fixée à n chiffres binaires. $x \in [0,1[^n$, les décimales de chaque composante sont les colonnes.
La "fitness" dont on cherche le minimum zéro est simplement la distance de Hamming d'avec la solution.
Les dessins testés ne montrent pas de différence significative entre eux, par contre les stratégies employées, grossièrement classées ci-dessous sont un peu chaotiques et se distinguent en faveur de "ssga" aléatoire ou avec tri et duplication des opérateurs avec élimination.

```
let decimales r n = let vd = (make_vect n 0) and b = ref r in
                    (* 0.001 indispensable à cause erreurs du float_of_int *)
          for e = 0 to n - 1 do b := 10. *. (!b); vd.(e) <- int_of_float (!b +. 0.001);
                                b := !b -. (float_of_int vd.(e)) done;
          vd;;

let dessin x = let n=vect_length x in let vb=make_vect n (make_vect n 0) in
          for i = 0 to n-1 do vb.(i) <- decimales x.(i) n done; vb;;

decimales 0.456123789 12 ☞ [|4; 5; 6; 1; 2; 3; 7; 8; 9; 0; 0; 0|]
dessin [|0.1234; 0.375; 0.20587|] ☞ [|[|1; 2; 3|]; [|3; 7; 5|]; [|2; 0; 5|]|]

let theta x = let vb = dessin x and n = vect_length x in let th = make_matrix n n 0 in
                    (* définition de la figure *)
          for i = 0 to n-1 do th.(0).(i), th.(n-1).(i), th.(i).(0), th.(i).(n-1), th.(n/2).(i) <- 1,1,1,1,1 done;
                    (* dist de Hamming *)
          let s = ref 0 in
          for i=0 to n -  1 do for j = 0 to n - 1 do s:= !s + (abs (th.(i).(j) -  vb.(i).(j))) done done;
          float_of_int (!s);;

for m = 0 to 6 do nl(); print_int m; print_string " -> "; for n = 2 to 10 do nl();
print_string " dim =" ; print_int n; print_string " ";
print_int (averag 12 500000 0.1 theta 0.0 0.99 n 35 33 m 5) done done;;
```

La reconnaissance d'une figure même 7*7 comme ce "H" est rapidement assez difficile puisqu'il faut de l'ordre de 50000 évaluations de la fonction distance avant qu'elle soit annulée. Ci-contre, dimensions du carré de 2 à 8.





15. Méta-algorithme pour trouver les meilleurs paramètres de la fonction phi

On peut utiliser l'algorithme "phi" avec ses meilleurs paramètres expérimentaux, comme méta-algorithme pour trouver les meilleurs triplets (mu, taux, clr), et ceci, sur un certain nombre de fonctions difficiles en dimensions variées.
Nous définissons pour cela une fonction "graph" chargée de dessiner trois projections du nuage des points (mu, taux, clr).

```
let graph t n m a b = let m' = round m and dim = vect_length t.(0).genes in
        efface (3 * m'); cadre m'; cadre (2 * m'); cadre (3 * m'); set_color red;
        for i = 0 to n - 1 do point t.(i).genes done
        (*  3 projections pour tableau t de points *)

where point p = let x, y, z, t =   (dilat m (contract p.(0) a b) a b),
                                   (dilat m (contract p.(1) a b) a b),
                                   (if dim > 2 then dilat m (contract p.(2) a b) a b else 0),
                                   (if dim > 3 then dilat m (contract p.(3) a b) a b else 0)
        in fill_rect x y 1 1;
        if dim > 2 then fill_rect (x + m') z 1 1;
        if dim > 3 then fill_rect (y + 2*m') t 1 1;;
        (* réalise les projections sur xy, xz et yt *)
```

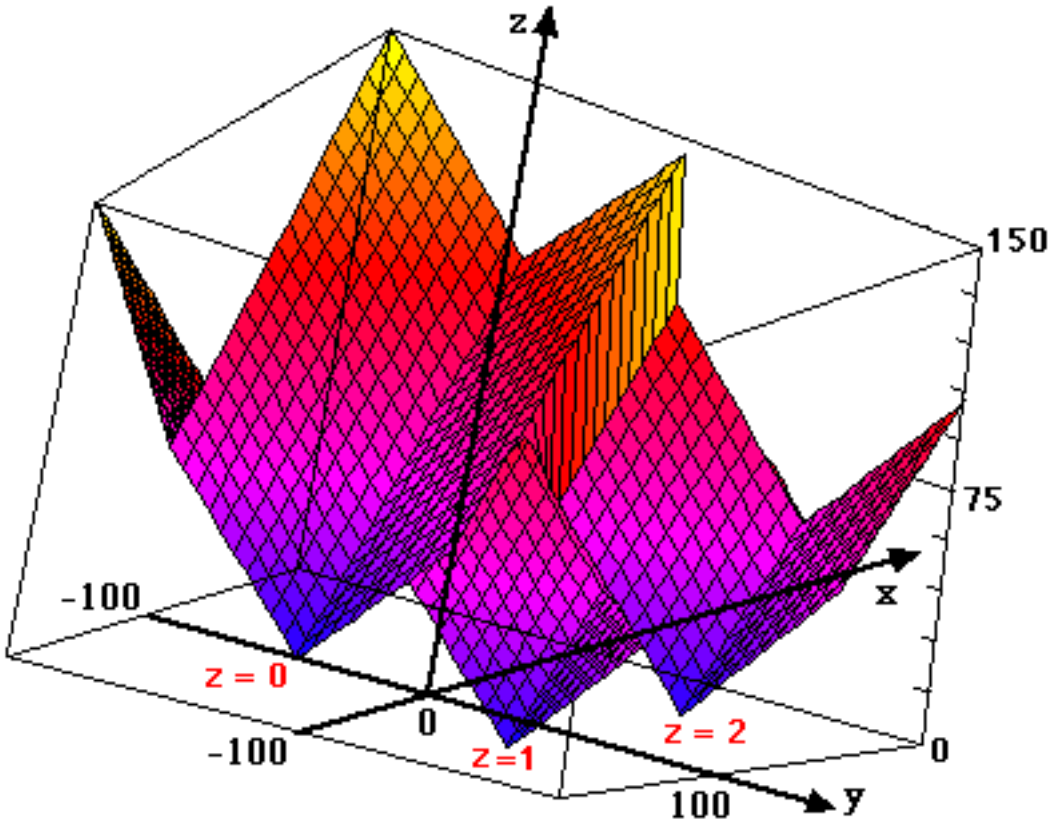
Ici, ce que nous voulons évaluer, ce sont des stratégies représentées par des triplets, et pour les évaluer, nous le faisons sur des fonctions difficiles, voire des cocktails de problèmes divers.

Soit, par exemple, la fonction "tripod" sur $[-100, 100]^2$ difficile à cause de ses bassins d'attraction (figure ci-dessous).

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

Facilement déclarée :

```
let tripod x = if x.(1) <. 0. then abs_float x.(0) +. abs_float (x.(1) +. 50.)
        else if x.(0) <. 0. then 1. +. abs_float (x.(0) +. 50.) +. abs_float (x.(1) -. 50.)
        else 2. +. abs_float (x.(0) -. 50.) +. abs_float (x.(1) -. 50.);;
```



On a vu en 10 que "averag mu max eps f a b dim taux clr m n" réalise la moyenne sur n tirages de "phi", donnant le nombre d'évaluations (majoré par max) de f sur $[a, b]^{dim}$, suivant une stratégie ssga avec une population de taile μ de taux τ, coefficient d'élimination clr, méthode m, pour atteindre un seuil ε.

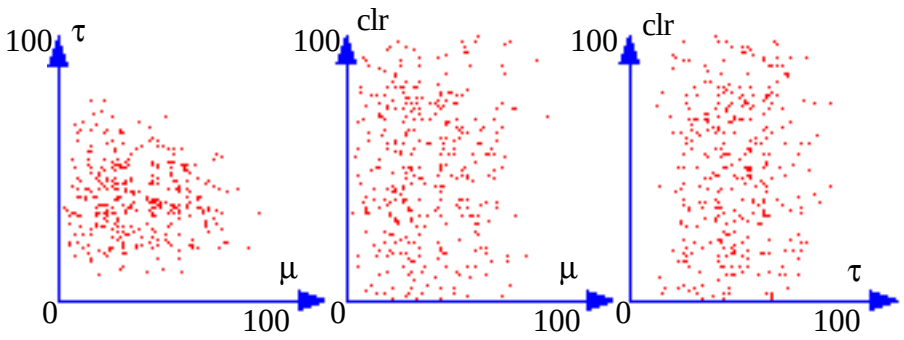
Les moyennes de différentes fonctions étant calculées sur 20 runs par exemple, il est possible de panacher les problème :

```
let test n x = float_of_int (
        (averag n (round x.(0))1e-04 parab (-100.) 100. 2 (round x.(1)) (round x.(2)) 3 20)
        + (averag n (round x.(0))1e-04 tripod (-100.) 100. 2 (round x.(1)) (round x.(2)) 3 20)
        + (averag n (round x.(0)) 1e-05 ros (-500.) 500. 2 (round x.(1)) (round x.(2)) 3 20) ) /.3.;;

let facil = test 5
and difficil = test 20
and encorpludificil x = float_of_int (
        averag 9 (round x.(0)) 1e-06 ros (-100.) 100. 2 (round x.(1)) (round x.(2))  3 2 0
        + averag 9 (round x.(0)) 1e-04 gri (-500.) 500. 10 (round x.(1)) (round x.(2)) 3 20);;
```

Les trois paramètres cherché sont situé dans [0, 100] et la dimension de ces fonctions est trois. On peut donc se servir de "phi" avec un grand nombre de points (par exemple 500) pour mieux voir la position du nuage. On lancera par exemple :

```
phi 5 100000 0. facil 0. 100. 3 33 33 3 true;;
phi 100 100000 0. encorpludificil 1. 100. 3 33 66 3 true;;
```

Ci-dessous, population de 500 individus testés chacun sur 20 essais de Rosenbrock et Tripod en dimension 2 et de Rastrigin et Griewank en dimension 30 (une génération par semaine de temps de calcul !) :



La première projection représente l'abscisse mu et l'ordonnée taux des 500 individus, la seconde (mu, clr) et la troisième (taux, clr), après 30 générations le meilleur triplet est (5, 33, 67). L'intérêt de ces expériences est que quelquesoit les problèmes considérés, s'il n'y a pas de réponse générale, on voit quand même presque toujours le nuage se former pour de petites valeurs de μ, entre 7 et 12 avec un taux de renouvellement compris entre 25% et 50%. Le troisième paramètre clr semble lui, moins pertinent.

16. Algorithme du recuit simulé

Cette méthode consiste à obtenir par itérations successives le minimum absolu d'une fonction, elle est inspiré d'une technique de refroidissement consistant à accepter dans certains cas une remontée de la fonction (pour ne pas descendre trop vite vers un minimum local).
En thermodynamique la probabilité d'avoir une augmentation d'énergie ΔE (la fonction que l'on veut minimiser suivant la loi de Boltzmann) est $e^{-\Delta E/\theta}$. On accepte un état voisin s_1 augmentant la fonction, dans la mesure où la probabilité ci-dessus est décroissante suivant ΔE. Le paramètre de contrôle (la température) va décroître, ce qui fait que pour un même ΔE, la probabilité d'accepter une remontée diminue suivant le refroidissement.
Il vaut mieux se concentrer sur des petites mutations, ainsi th = 10 adapté à Griewank 10^5 à Rastrigin. La difficulté de cette méthode, qui, par ailleurs, donne de bons résultats, réside dans la détermination des paramètres. On peut montrer la convergence sous certaines conditions reliant n et la loi de décroissance de la température. La température initiale peut être déterminée par une probabilité, par exemple 1/2, d'accepter une hausse au cours du premier palier, si m est la valeur moyenne de ces hausses, alors $\theta_0 = m / \ln(2)$. Le nombre d'itérations sur chaque palier peut être de l'ordre de la centaine, la loi de décroissance de la température est généralement prise comme une suite géométrique telle que $\theta_{n+1} = 0{,}9\ \theta_n$ Plus précisément :

1 Choisir un état initial s_0
2 Faire le «palier» consistant à répéter n fois
Chercher un voisin s_1 de s_0, on calcule $\Delta = f(s_1) - f(s_0)$
Si $\Delta < 0$ alors $s_0 \leftarrow s_1$
Si $\Delta > 0$ alors on accepte la même affectation de s_0 avec une probabilité $e^{-\Delta/\theta}$
3 On abaisse la température θ et on réitère ces paliers jusqu'au test d'arrêt.

```
let recuit max eps f a b dim n = let tp = make_vect 2 {score = 0.; genes = make_vect dim []}
        and nv = ref 1 and th = ref 100. in
        let init = applic tp 0 1 f a b 3 9  in
                (* création du point de départ grâce à l'opérateur migtout numéro 9 *)
while !nv < max & eps < tp.(0).score do palier n; th := 0.9*.(!th);  nl();
        print_string (" nv = " ^ (string_of_int !nv) ); vue tp a b done; !nv
        where palier n = for i = 1 to n do tp.(1) <- tp.(0);
                let v0 = tp.(0).score and mieux = applic tp 0 1 f a b 3 (random__int 10) in
                let v1 = tp.(0).score in incr nv;
                if not mieux && (random__float 1. > exp ((v0 -. v1) /. (!th)))
                then tp.(0) <-  tp.(1) (* on ne modifie pas le point courant *)
done;;
```

Ici, nous définissons le voisin comme le résultat de l'application d'un opérateur génétique parmi les opérateurs unaires, ceci étant parfaitement modifiable.

Mais il n'est pas besoin de définir les voisins, simplement, la fonction "applic" pour le point courant, ira chercher au hasard un opérateur parmi les 10 premiers, c'est à dire unaire.
La plongée de cet algorithme dans le bassin d'attraction d'un minimum local est presque inévitable.
On lancera par exemple :

```
recuit 10000  10e-4 tripod (-100.) 100. 2 10;;
```

17. Algorithme du "Hill climbing"

Cette méthode locale, partant d'un point choisi aléatoirement, consiste itérativement, à examiner tout le voisinage du point courant afin de se placer au meilleur d'entre eux pour l'étape suivante. Naturellement, en chaque point x on définit un voisinage fini V_x de points pour lesquels on peut calculer f.
Ce peut être des points voisins au sens d'une distance discrète ou bien, comme on le verra plus tard, des points définis à partir de x par des opérateurs génétiques.

1 Choisir x aléatoire dans le domaine de la fonction f à minimiser
2 $t \leftarrow 0$ $f_{min} \leftarrow +\infty$
3 incr(t)
4 Evaluer f sur chacun des points voisins de x, soit x' tel que $f(x') = \min \{f / V_x\}$
5 Si $f(x') < f(x)$ alors $x_{min} \leftarrow x'$
6 $x \leftarrow x'$
7 Si $t < t_{max}$ alors aller en 3

L'inconvénient manifeste de ces méthodes locales est que l'on ne peut pas vraiment espérer sortir d'un minimum local, cependant, les résultats sont meilleurs qu'avec le recuit simulé pour la raison qu'on utilise toutes les heuristiques d'exploration de l'espace grâce aux opérateurs unaires. Ces 10 opérateurs unaires du 6 sont encore nécessaires.

```
let hill max eps f a b dim = let tp = make_vect 12 {score = 0.; genes = make_vect dim []}
        and nv = ref 1 in let init = applic tp 0 1 f a b 3 9  in
                (* création du point de départ grâce à l'opérateur migtout numéro 9 *)
        while !nv < max & eps <. tp.(0).score do nl(); vue tp a b;
                for i = 1 to 10 do tp.(i) <- tp.(0) done;
                for i = 1 to 10 do if applic tp i 1 f a b 3 i then print_string "  mieux " done;
                nv := !nv + 10; triv tp 11
        done; nl(); vue tp a b ; !nv;;

hill 1000 10e-4 tripod (-100.) 100. 2;;
```

18. Algorithme de Nissen

Le principe de cet algorithme est de maintenir un ancêtre, celui qui a pu engendrer les autres durant une "ère", un un père "provisoire". L'exploration du voisinage est réalisé par des "mutations", petites variations au sein du codage des éventuelles solutions.
Les ères sont donc constituées par des suites de générations où il n'y a pas de progrès, et où l'ancêtre détient la meilleure solution. La convergence n'est pas monotone puisqu'il suffit d'une amélioration pour modifier l'ancêtre.

Cependant si λ est suffisamment grand, au risque de tomber plusieurs fois dans le même bassin d'attraction, on passe d'un minimum local à un autre. En fait, c'est l'éventail des possibilités d'exploration grâce aux λ enfants (ici on prend 10) qui est le trait le plus important. La déstabilisation est là pour éviter la convergence prématurée, elle consiste à changer d'ère chaque

fois qu'un fils est meilleur que son père ou que le nombre de génération est jugé suffisant, en ce dernier cas, cela revient à changer de point initial.

1 Engendrer λ individus (par des mutations, transposition, ... mais pas de croisement).
2 Soit I_0 le meilleur d'entre eux (l'ancêtre).
3 $I_1 \leftarrow I_0$ (I_1 père provisoire) ère $\leftarrow$ 0 t $\leftarrow$ 0
4 répéter incr (t) (nouvelle génération)
si ère = $ère_{max}$ alors ère $\leftarrow$ 0 (déstabilisation)
Engendrer λ individus à partir du père I_1, soit I_2 le meilleur d'entre eux
$I_1 \leftarrow I_2$ (le meilleur fils prend la place du père à chaque génération)
si I_2 meilleur que I_1 alors ère $\leftarrow$ 0; $I_0 \leftarrow I_2$ (nouvel ancêtre)
sinon incrémenter (ère)
jusqu'à t = t_{max}

Voir : Nissen V.A. *A new efficient evolutionary algorithm for quadratic assignment problem,* Operations research proc. of the 21th annual meeting of DGOR p259-269, 1993

```
let nissen max eps f a b dim eremax  lbd = let nv = ref 1 and  ng = ref 0 and
        tp = make_vect (1 + lbd) {score = 0.; genes = make_vect dim []}  in aff oper operef no;
        let init = applic tp 0 1 f a b 3 9 in for i = 1 to lbd do tp.(i) <- tp.(0) done;
                (* tp(0) point de départ grâce à l'opérateur de numéro 9 *)
        let anc = ref tp.(0) and ere = ref 0 and mieux = ref false in (* ancêtre *)
while !nv < max & eps <. tp.(0).score do (* tp.(0) est le père *)
        if !ere = eremax then (let init = applic tp 0 1 f a b 1 9 in ere := 0);
        let pere = ref tp.(0) in for i = 1 to lbd -1 do mieux := applic tp i 1 f a b 3 i done;
        triv tp (1 + lbd); incr ng;
        if tp.(0).score < (!pere).score (* le meilleur des enfants est meilleur que le père *)
        then (ere := 0; anc := tp.(0); nl(); print_string "Nouv. ère ") else incr ere;
        nl(); print_string ("ng " ^ (string_of_int !ng) ^ " nv = " ^ (string_of_int !nv) ); vue tp a b;
done; !nv;;
```

```
(* on lancera par exemple *)
nissen 1000 10e-4 gri (-100.) 100. 2 100 9;;
```

Quelques résultats (ES7 est la stratégie définie en 8) :

	Hill-climbing	Nissen	Recuit simulé	GA 0.5-0.1	ES 7	SSGA 10%	SSGA 25%	SSGA 33%	SSGA 50%
De Jong dim 3	667	2881	5321	13256	2286	1460	1132	1254	1307
Paraboloïde dim 3	1602	6275	28251	max	5396	6155	4134	4056	5158
... ... dim=10	7433	17129	31721	max	47391	16100	13842	12560	13827
Rosenbrock dim 2	29010	44090	47816	max	31952	39431	33359	32587	40210
Rastrigin dim 10	13449	max	23626	max	max	15422	13651	13950	15564
Griewank dim 10	max	max	max	max	max	73298	70819	43781	45450

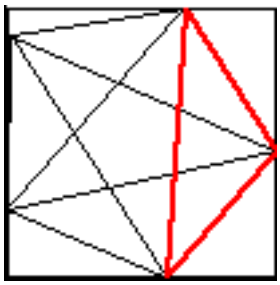
Moyennes des nombres d'évaluations pour obtenir une contrainte (meilleure valeur < eps et nombre d'évaluations < max) sur 20 essais aléatoires. Pour le recuit simulé, le paramètre initial de température n'est pas aisé à fixer, il est fonction des valeurs que peut prendre la fonction sur le domaine retenu. En grisé les deux meilleurs résultats par ligne.

19. Application au problème de Heilbronn

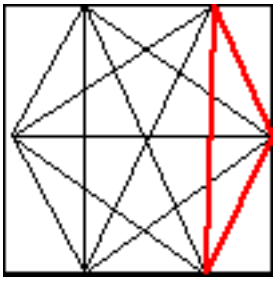
Soient n points sur $[0, 1]^2$, en traçant tous les triangles possibles formés avec ces points, on retient à chaque fois celui de plus petite aire, et on cherche à la maximiser, on définit :
f(n) = max{min aire {triangles possibles}}. On code en dimension 2n sur [0, 1[un chromosome étant $x = (x_0, ... x_{2n-1})$ où le point M_k a pour coord. x.(2k) et x.(2k+1).
Pour n = 3 ou 4 f(n) = 1/2 trivialement, f(4) = 0,499993331758 obtenu, mais après ces valeurs, la conjecture est que $1/n^{8/7} < f(n) < 1/n^2$.

```
let aire xa ya xb yb xc yc = 0.5*.abs_float(xb*.yc-.xb*.ya-.xa*.yc-.xc*.yb+.xc*.ya+.xa*.yb);;
let np = dim /2;; (* nombre de points *)

let hb x = let np = (vect_length x) / 2  and m = ref 1. and a = ref 0 and b = ref 0 and c = ref 0 in
        for i = 0 to np-3 do for j = i+1 to np-2 do for k = j+1 to np-1 do
                let s = aire x.(2 * i) x.(2 * i + 1) x.(2 * j) x.(2 * j + 1) x.(2 * k) x.(2 * k + 1) in
                        if s < !m then (m := s; a := i; b := j; c := k) done done done;
        (1.-. !m);;
        (* hb renvoie 1 - l'aire minimale *)

let graphe tp  =
        set_color black; moveto 100 0; lineto 0 0; lineto 0 100; lineto 100 100;  lineto 100 0;
        let np = (vect_length tp) / 2 and m = ref 1. and a = ref 0 and b = ref 0 and c = ref 0
        and x = map_vect (fun l -> contract l 0. 1.) tp.(0).genes in
        let pt i = moveto (round(100. *. x.(2 * i))) (round(100.*.x.(2 * i + 1)))
        and ln i = lineto (round (100. *. x.(2 * i))) (round(100.*.x.(2 * i + 1)))
        in      for i = 0 to np-3 do for j = i+1 to np-2 do for k = j+1 to np-1 do
        let s = aire x.(2* i) x.(2* i + 1) x.(2* j) x.(2 * j + 1) x.(2* k) x.(2* k + 1) in pt i; ln j; ln k; ln i;
                if s < !m then (m := s; a := i; b := j; c := k) done done done;
                set_line_width 2; set_color red; pt (!a); ln(!b); ln (!c); ln (!a); moveto 10 110;
                draw_string ("Max aire min = " ^ (string_of_float !m)); read_key(); close_graph();;

phi 9 1000 0. hb 0. 1. 16 33 33 3 true;;
```

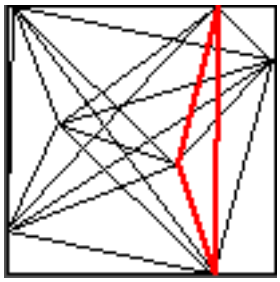
Ce problème n'est pas supervisé, on ne connait pas le min, on place donc eps = 0. et un grand nombre max, pour 4 points, dim = 8. Il reste une petite modification à faire dans la fonction afin qu'elle retourne le taleau tp, ou bien qu'elle appelle la fonction "graphe".



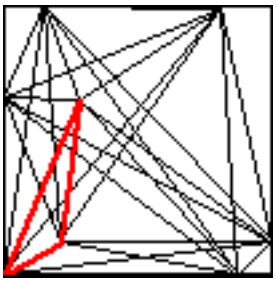
5 points f(5) = 0,19003586915



f(6) = 0,116342699369 (but 1/8)



f(7) = 0,076627646264



f(8) = 0,0491038762498