Les algorithmes génétiques

—

TIPE de l'année 2001-2002
Lycée Joffre Montpellier

Julien BOURREE - Benoit ISAAC
tipe@helicos.com

27 juin 2002

Table des matières

Introduction

Les algorithmes génétiques trouvent leurs origines dans les années 1960, lorsque des biologistes ont tenté de simuler, grâce aux ordinateurs, des systèmes biologiques. Mais ce sont principalement les travaux de John Holland, dans les années 1970, qui aboutiront aux algorithmes génétiques tels que nous les connaissons aujourd'hui.

Les algorithmes génétiques sont des algorithmes d'optimisation s'appuyant sur les techniques de l'Evolution naturelle, décrites par Darwin. Ayant constaté que les êtres complexes qui vivent aujourd'hui sur Terre sont dûs principalement au hasard et à la sélection naturelle, des chercheurs ont tenté de reproduire l'Evolution de manière informatique : l'algorithmique génétique était née.

En décomposant et recomposant une sorte de «brin d'ADN», les algorithmes génétiques peuvent optimiser des fonctions comportant de nombreuses variables, pour lesquelles il n'existe aucune technique rapide d'optimisation. En effet ils permettent d'obtenir très rapidement de très bonnes solutions approchées, et c'est ce que nous allons voir à travers un exemple simple, pour lequel nous créerons un algorithme génétique complet (codé en langage Caml).

1 L'exemple choisi : calcul de plus court temps de parcours

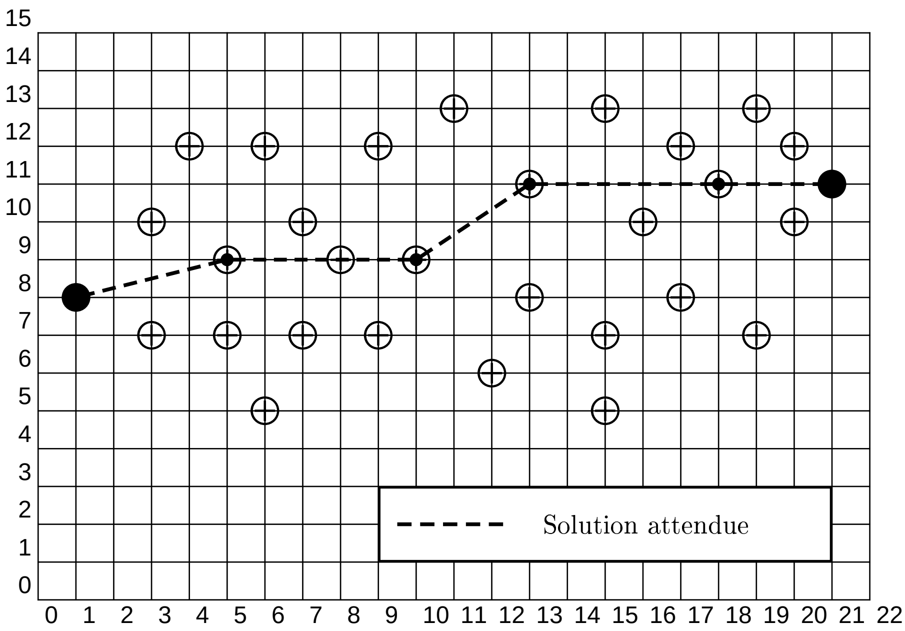
Nous étudierons ici un exemple simple, pour lequel nous pouvons obtenir (par d'autres techniques) la solution optimale : la minimisation d'un temps de parcours. Considérons un pilote d'avion qui souhaite rallier deux aérodromes. Connaissant les caractéristiques de son appareil (vitesse, autonomie) et la carte de la région qu'il va survoler (notamment l'emplacement de tous les aérodromes intermédiaires), il souhaite minimiser le temps de vol entre les deux aérodromes, ravitaillements compris.

Nous choisirons ici les caractéristiques suivantes :
- Vitesse de l'avion : 100 km.h^{-1}
- Autonomie de l'avion : 600 km
- Durée d'un ravitaillement : 45 mn (soit 0.75 h)

La figure 1 représente la carte de la région : les aérodromes de départ (à gauche) et d'arrivée (à droite) sont représentés par des disques pleins, les aérodromes intermédiaires (au nombre de 28), par des cercles. L'échelle est la suivante :
- Le côté d'une case représente 100 km.

Le nombre d'aérodromes intermédiaires étant faible, nous pouvons trouver la solution "à la main" : c'est celle représentée en pointillés.

Fig. 1 – Carte des aérodromes



Ici l'exemple est simple, pour que nous puissions savoir si l'algorithme se rapproche de la solution optimale : en pratique, les algorithmes génétiques sont utilisés, nous l'avons dit, pour des problèmes complexes, pour lesquelles il n'existe pas de méthode rapide de résolution (imaginons un millier d'aérodromes, par exemple !).

2 Fonctionnement des algorithmes génétiques

2.1 Principe général

La nature, à partir d'un brin d'ADN très simple (4 «briques» seulement), a réussi à créer des êtres très complexes et totalement adaptés à leur environnement. Selon Darwin, c'est le principe de la sélection naturelle qui a permis aux individus les plus adaptés de se reproduire et de survivre, jusqu'à aboutir aux êtres que nous connaissons aujourd'hui. A partir de ce constat, les chercheurs ont décidé d'appliquer les principes de l'évolution naturelle à des problèmes dont ils ne connaissent pas la solution, souvent car le nombre de variables est trop élevé pour pouvoir tester toutes les possibilités.

De la même façon que la nature, au fur et à mesure des générations, optimise la fonction «survie», les algorithmes génétiques, en appliquant les principes de fonctionnement de la nature, vont optimiser des fonctions qui peuvent être très complexes, et cela de manière très rapide grâce à l'utilisation de l'ordinateur.

Fig. 2 – Analogie avec la nature

Nature	**Algorithme génétique**
individu	solution à un problème
population	ensemble de solutions
but : survivre	but : optimiser une fonction
adéquation au milieu	qualité de la solution (**fitness**)
brin d'ADN	représentation codée d'une solution
mutation	modification d'une solution
crossover	décomposition et recombinaison
sélection naturelle	roulette biaisée (cf 2.2.3)

A partir d'une population initiale générée aléatoirement, et au gré des décompositions / recombinaisons des brins d'ADN codant les individus de cette population, les algorithmes génétiques vont pouvoir donner très rapidement de très bonne solutions approchées aux problèmes d'optimisation complexes.

Le principe de fonctionnement de tout algorithme génétique peut être représenté suivant le schéma suivant :

Principe de fonctionnement :

- Création de la population initiale
- **Tant que** (condition d'arrêt non vérifiée)
  Faire
  - Evaluer chaque individu
  - Sélectionner les individus
  - Appliquer les opérateurs choisis

  FinFaire

où la condition d'arrêt peut être par exemple un nombre maximum de générations, ou encore un seuil suffisant pour la fonction à optimiser : si l'algorithme génétique a trouvé une solution «assez» bonne (au sens du seuil précédent) ou si le nombre de générations maximum a été atteint, l'algorithme s'arrête. En effet pour des raisons pratiques de temps de calcul, il est nécessaire de donner une condition d'arrêt à l'algorithme, car il ne peut s'arrêter tout seul, puisqu'il n'a aucun moyen de «savoir» si la solution qu'il a trouvée est optimale (les algorithmes génétiques sont principalement utilisés, rappelons-le, pour optimiser des fonctions dont on ne connaît pas la solution optimale !)...

2.2 Détails des étapes

Un algorithme génétique recherche donc le ou les extrema d'une fonction définie sur un espace de solutions.

Pour pouvoir utiliser un algorithme génétique, on doit disposer des 5 éléments suivants :

- Un principe de codage des solutions potentielles (pour la nature, c'est l'ADN).
- Un moyen de générer une population initiale, répartie de façon assez homogène sur tout l'espace de solutions.
- Une fonction à optimiser, qui servira à évaluer chaque individu de la population. La qualité d'un individu dépendra donc de son aptitude à résoudre le problème posé (suivant la fonction à optimiser choisie), ce que l'on appellera par la suite sa **fitness**, car le mot anglais correspond bien à cette définition.
- Des opérateurs permettant de diversifier la population au cours des générations, afin d'explorer l'espace de données.
- Des paramètres comme le nombre d'individus dans la population, les probabilités pour un individu de subir un crossover ou une mutation...

Ici nous opterons pour les caractéristiques suivantes :

- Principe de codage : **Vecteur de bits (0 ou 1)**.
- Population initiale : **générée aléatoirement**.
- Fonction à optimiser : **temps de parcours (à minimiser)**.
- Opérateurs : **mutation, crossover et sélection selon le principe de la roulette biaisée**.
- Paramètres : **variables selon les expériences**.

Remarque Au lieu de minimiser le temps de parcours, pour des raisons de commodité, nous chercherons à rendre maximale la différence entre un temps donné et le temps de parcours codé par l'individu (ce qui revient au même, mais nous permet d'écrire un algorithme génétique qui cherche les maxima d'une fonction, ce qui est plus simple pour la suite).

Les individus seront donc représentés par des vecteurs de 0 et de 1, et nous appellerons **gène** chaque 0 ou 1 .

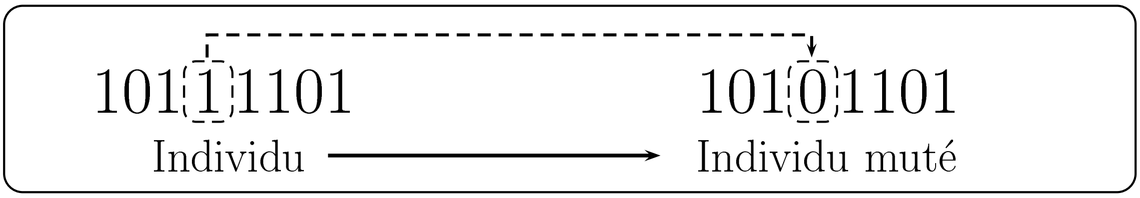
2.2.1 Mutation

La mutation est un opérateur permettant d'introduire de la diversité dans la population. Par analogie avec la nature, elle consiste à remplacer un gène par un autre aléatoirement (avec une probabilité définie à l'avance). Ici donc, la mutation remplacera un 0 par un 1 et inversement.

Voir figure 3, page suivante

L'opérateur mutation permet d'**explorer** l'espace des solutions, en introduisant aléatoirement des variations dans les individus.

Fig. 3 – Mutation

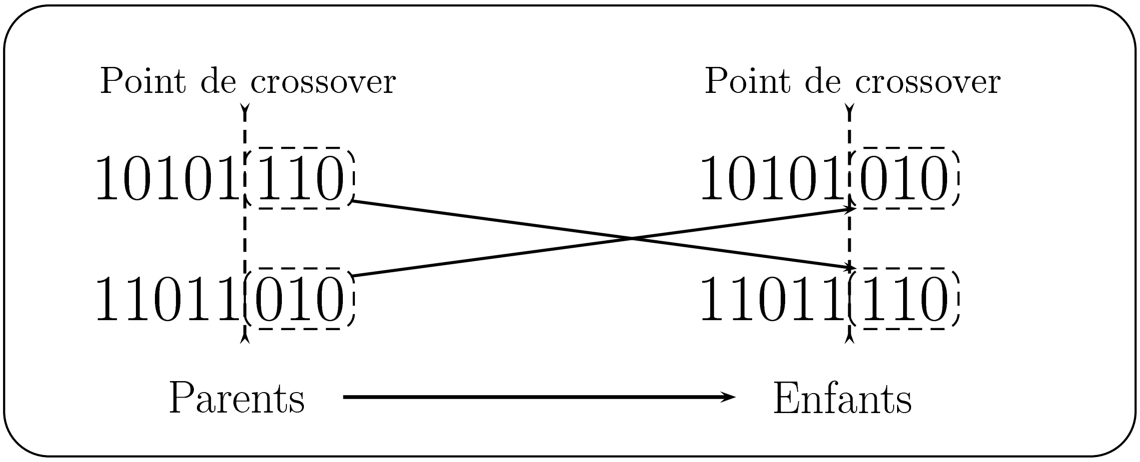


2.2.2 Crossover

Le crossover est un opérateur ayant pour but de recombiner deux bonnes solutions en espérant obtenir une solution encore meilleure.

Pour cela, l'algorithme génétique décompose deux individus (les «parents») et les recombine pour obtenir deux nouveaux individus (les «enfants»).

Fig. 4 – Crossover (à un point)



Dans le crossover à un point, représenté figure 4, l'algorithme génétique détermine (aléatoirement) un **point de crossover** : les individus sont «découpés» de part et d'autre de ce point, et recombinés.

Il existe aussi le crossover à deux points, qui fonctionne sur le même principe mais en deux endroits différents ; enfin il existe aussi le crossover uniforme, qui lui scinde les individus en un nombre aléatoire de points.

L'opérateur crossover permet, lui, d'**exploiter** les solutions déjà existantes dans la population.

2.2.3 Sélection

Afin de «diriger» l'algorithme vers les régions de l'espace de solutions contenant les meilleurs individus, une sélection sera opérée d'une génération à l'autre, un peu à la manière de la sélection naturelle, qui laisse survivre et se reproduire les individus les plus adaptés à leur milieu.

Elle se fera selon le principe de la **roulette biaisée**. Ce terme s'explique ainsi :

– **roulette** : la sélection sera aléatoire.

- **biaisée** : les individus n'auront pas tous la même probabilité de survivre et de se reproduire.

Pour cela, nous adopterons la méthode suivante :

1. Nous calculerons la somme de toutes les fitness de tous les individus de la population.
2. Nous pourrons alors calculer la fitness relative (**rfitness**) de chaque individu, celle-ci étant le rapport de sa fitness sur la somme totale de toutes les fitness, calculée en 1.
3. Puis chaque individu aura une probabilité égale à sa fitness relative d'être sélectionné, voire de se reproduire.

Rappelons que nous utilisons ici un algorithme génétique qui va chercher les maxima de la fonction à optimiser (cf remarque page 6). Ainsi la fitness relative est bien d'autant plus grande que l'individu est «bon» (au sens du problème choisi).

Prenons un exemple simple (avec une fonction d'évaluation quelconque), afin d'expliciter cette méthode :

La population est de **3 individus**, comptant chacun **8 gènes**.
- L'individu 1 est : $I_1 = (11010101)$, il a une fitness de 12.
- L'individu 2 est : $I_2 = (10011110)$, il a une fitness de 72.
- L'individu 3 est : $I_3 = (10010010)$, il a une fitness de 36.

La somme des fitness est de 120, et les **fitness relatives** sont donc :
- Individu 1 : **0.1** ;
- Individu 2 : **0.6** ;
- Individu 3 : **0.3**.

Nous affectons alors à chaque individu une fitness cumulative (**cfitness**), qui correspond à la somme des fitness relatives de tous les individus situés avant l'individu donné, en y ajoutant sa propre fitness relative.
Voici donc les **fitness cumulatives** :
- Individu 1 : **0.1** ;
- Individu 2 : **0.7** ;
- Individu 3 : **1.0**.

De manière imagée, chaque individu se voit ainsi affectée d'une «marche d'escalier», d'autant plus haute que sa fitness relative est bonne, donc que l'individu est bon (par rapport au reste de la population). A partir de cet escalier, nous allons pouvoir choisir quels individus seront sélectionnés (voire reproduits) dans la génération suivante.

C'est ici qu'intervient le hasard. Pour garder une population de taille constante, l'algorithme génétique va alors générer 3 chiffres **aléatoirement**, situés entre 0 et 1, et reproduira l'individu qui possède la «marche d'escalier» correspondante.

Lors de notre test, l'algorithme a généré les 3 nombres aléatoires suivants :
- 0,32454322. Ce chiffre est situé entre 0.1 et 0.7, c'est l'individu 2 qui sera sélectionné.
- 0,57653529. Ce chiffre est situé entre 0.1 et 0.7, c'est l'individu 2 qui sera sélectionné.
- 0,84536217. Ce chiffre est situé entre 0.7 et 1.0, c'est l'individu 3 qui sera sélectionné.

Après cette sélection, la population est toujours composée de 3 individus : 2 fois l'individu 2 (qui a donc été dupliqué) et 1 fois l'individu 3. L'individu 1, dont la fitness relative était très faible, n'a pas survécu. Ce sont ces 3 individus qui subiront alors crossover et mutation, dans le but d'obtenir des solutions encore meilleures.

On voit que cette méthode permet, à chaque génération, d'augmenter la fitness moyenne de la population, et donc d'obtenir très rapidement de très bons individus. Le hasard entre en jeu pour que des individus même très «faibles» puissent parfois être sélectionnés, afin de conserver leur patrimoine génétique dans l'espoir qu'il serve un jour, lors d'un crossover, à améliorer une autre solution. En effet une solution très mauvaise peut par exemple être très bonne en ce qui concerne le début du parcours, alors que la fin du parcours est effectuée de manière catastrophique. Un crossover pourra alors «recoller» ce début de très bonne solution avec la fin d'une autre solution qui elle, pêcherait au début du parcours : on obtiendra alors au final un très bon individu.

2.3 Eléments de la théorie des algorithmes génétiques

Ces dernières lignes sur la roulette biaisée nous amènent à essayer de comprendre le fonctionnment théorique d'un algorithme génétique.

Afin d'expliquer celui-ci, introduisons quelques notions. **Nous considérons ici des individus codés avec 8 gènes, pour simplifier.**

Un «**schéma**» est construit en introduisant le symbole *indifférent* ($\star$) dans l'alphabet des gènes (qui ne contenait juqu'à présent que 0 et 1). Un schéma représente tous les individus lui correspondant sur tous les gènes *fixés*, c'est-à-dire autres que «$\star$» : un schéma représente donc en fait un sous-ensemble de l'espace des solutions.

Ainsi le schéma ($\star$1110010) code 2 individus :

{(11110010),(01110010)} sont deux représentants de ce schéma;

tandis que le schéma ($\star 11 \star 0010$) code, lui, 4 individus.

Nous voyons que plus le nombre d'étoiles dans un schéma est grand et plus le sous-ensemble qu'il représente est grand (au sens du cardinal).

Nous appellerons «**ordre**» d'un schéma S (noté $o(S)$) le nombre de gènes *fixés* dans celui-ci (c'est-à-dire 8 moins le nombre d'étoiles).

Nous appellerons «**longueur**» d'un schéma S (noté $\delta(S)$) la «distance» entre le premier et le dernier gène *fixés*.

Ainsi, si nous prenons comme exemple les schémas :

$$\begin{aligned} S_1 &= (\star\star\star 00 \star 1 \star) \\ S_2 &= (100 \star\star 10 \star) \end{aligned}$$

Nous avons donc :

$$\begin{aligned} o(S_1) &= 3 \quad et \quad \delta(S_1) = 7-4 = 3 \\ o(S_2) &= 5 \quad et \quad \delta(S_2) = 7-1 = 6 \end{aligned}$$

L'ordre définit en fait la «spécialisation» d'un schéma, et la longueur définit la «compacité» de l'information contenue dans un schéma. Remarquons qu'un schéma ne contenant qu'un seul bit fixé a une longueur de zéro.

Si l'on observe alors le fonctionnement d'un algorithme génétique simple (cf 2.1), on s'aperçoit que les opérations qui modifie la population au fur et à mesure des générations sont : la sélection, la mutation et le crossover. En effet l'évaluation n'a aucun effet sur la population.

Nous cherchons à calculer la probabilité de survie d'un schéma donné. Celle-ci dépend des deux caractéristiques définies plus haut :

1. **L'ordre** : Plus l'ordre d'un schéma est grand, et plus la mutation a de chances de le détruire. En effet plus le nombre de gènes fixés est grand et plus la mutation a de chances d'opérer sur un de ces gènes fixés.

2. **La longueur** : Plus la longueur d'un schéma est grande, et plus le crossover a de chances de le détruire. En effet plus la distance entre le premier et le dernier gène fixés est grande et plus le point de crossover a de chances d'être choisi entre ces deux gènes fixés, «cassant» ainsi le schéma étudié.

La probabilité de survie est donc d'autant plus forte que le schéma est court et que son ordre est faible.

Enfin, il est clair que **la roulette biaisée**, du fait de son fonctionnement, va favoriser les schémas qui codent des individus dont la fitness est supérieure à la moyenne. Ainsi dans notre exemple, la moyenne des fitness des individus est de : $\frac{120}{3} = 40$. L'individu **2**, qui a une fitness de 72, a donc une fitness $\frac{72}{40} = 1,8$ fois plus élevée que la moyenne. On peut donc espérer qu'il sera reproduit $1,8$ fois (c'est-à-dire 2 fois, probablement) dans la génération suivante. Et c'est ce qui se passe effectivement : dans la génération suivante, l'individu **2** est dupliqué, il a bien été recopié 2 fois.

Si l'individu **2** appartient à une région de l'espace des solutions codée par un schéma S (ne codant pas les deux autres individus), par exemple : $S = (\star\star\star 1111\star)$, on peut légitimement penser que ce schéma va donc avoir plusieurs représentants dans les générations suivantes (dans la génération 1, il avait un seul représentant, dans la génération 2, il en a déjà 2 fois plus). Si l'on suppose que dans les générations suivantes, il code toujours des individus dont la fitness reste au-dessus de la moyenne avec le même ratio $(1,8)$, le nombre d'individus codés par ce schéma va donc croître exponentiellement.

Ceci est vrai dans le cas où le schéma arrive à survivre à la mutation et au crossover. Cela nous amène au «théorème» suivant :

Théorème des schémas : Les schémas courts (au sens de la longueur définie plus haut), d'ordre faible, codant des individus situés au-dessus de la moyenne, sont représentés par un nombre d'individus croissant exponentiellement au fur et à mesure des générations créées par un algorithme génétique.

Un résultat immédiat suit ce théorème : l'algorithme génétique explore l'espace des solutions en privilégiant les régions codées par des schémas «au-dessus de la moyenne», courts et d'ordre faible. C'est à partir de ces schémas que les crossover vont effectuer les échanges d'information entre les individus :

Hypothèse des blocs de construction : Un algorithme génétique cherche des solutions quasi-optimales obtenues par juxtaposition de schémas «très bons», courts et d'ordre faible, appelés **blocs de construction**.

Mais ceci n'est qu'une hypothèse, formulée à partir des différents exemples pratiques étudiés par les chercheurs. Le comportement d'un algorithme génétique étant en partie dû au hasard, il est en effet difficile de *démontrer* cette hypothèse.

Nous allons voir qu'elle s'applique parfaitement à notre problème.

3 Codage de l'exemple

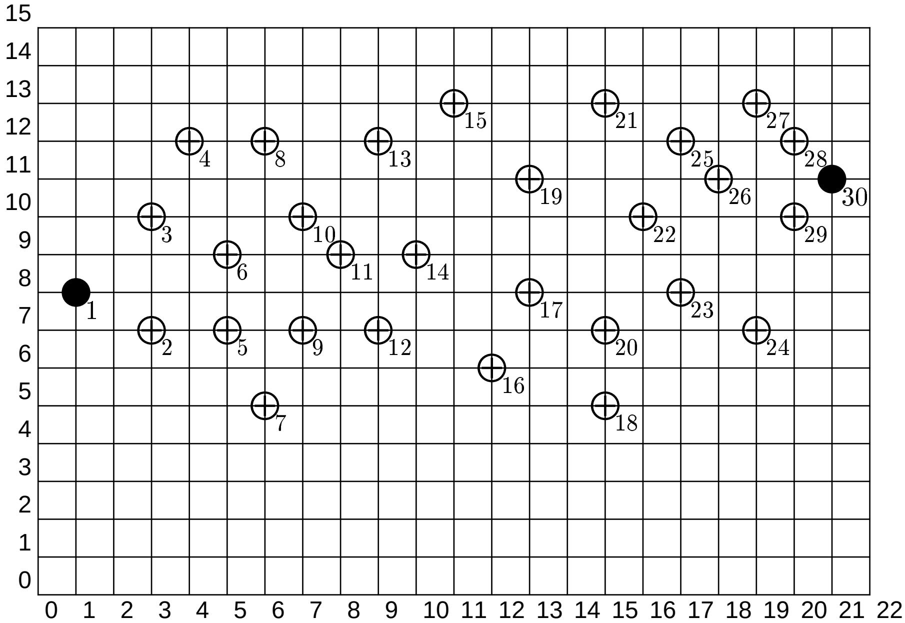
Il faut maintenant coder l'exemple, afin de voir réellement "tourner" un algorithme génétique. Nous avons choisi d'utiliser pour cela le langage CAML, utilisé en option informatique, car il permet de coder des problèmes assez complexes de manière relativement simple.

3.1 Implémentation du problème

Le principe de codage

Afin d'appliquer la théorié étudiée, une **"solution" possible** au problème du pilote sera représentée par un vecteur composé de bits. Pour cela, nous avons décidé de numéroter les aérodromes par ordre décroissant d'éloignement à l'arrivée.

FIG. 5 – CARTE DES AÉRODROMES



Ainsi, le premier élément du vecteur code un possible ravitaillement dans l'aérodrome le plus éloigné de l'arrivée (en comptant l'aérodrome de départ) : s'il y a un "1" à cet emplacement, la solution code effectivement un ravitaillement sur cet aérodrome.

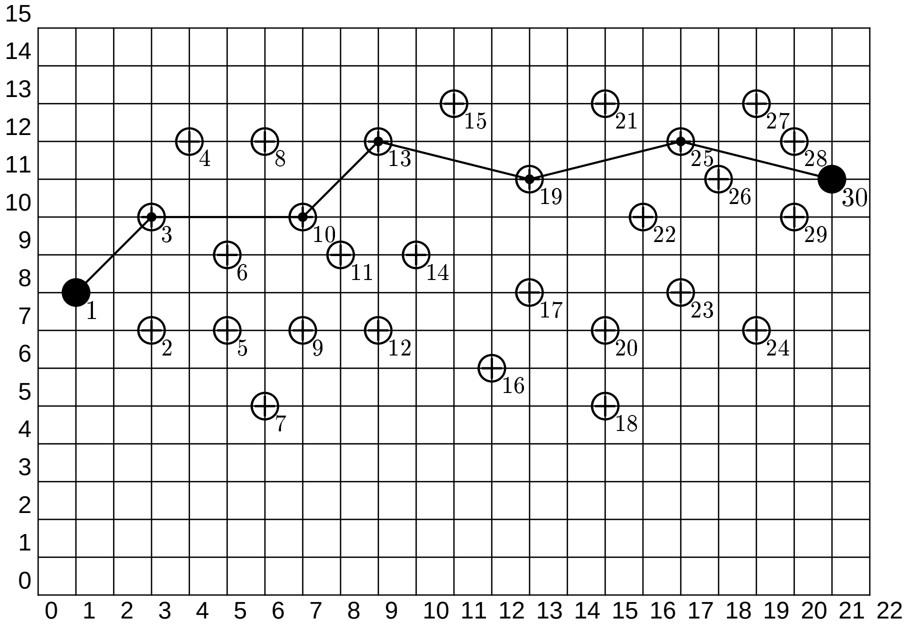
Par exemple, le vecteur :

```
[|0,1,0,0,0,0,0,0,1,0,0,1,0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,0|]
```

code un vol incluant des ravitaillements dans les aérodromes 3,10,13,19 et 25 :

Voir figure 6, page suivante

FIG. 6 – TRAJET [0,1,0,0,0,0,0,0,1,0,0,1,0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,0]



3.1.1 Les données

L'algorithme génétique devra prendre en argument : la matrice contenant les coordonnées des aérodromes, l'autonomie de l'appareil, sa vitesse, le temps nécessaire à chaque ravitaillement, le nombre d'individus par génération, les probabilités de crossover et de mutation.

3.1.2 La population

Les **"individus"** de la population seront caractérisés par le vecteur codant leur trajet (un vecteur d'entiers), leur "fitness" correspondant au résultat de la fonction d'évaluation (3.2), leur fitness relative et leur fitness cumulative (cf la roulette biaisée, 2.2.3), ces 3 fitness étant des réels :

```
type individu = { mutable trajet   : int vect ;
                  mutable fitness  : float    ;
                  mutable rfitness : float    ;
                  mutable cfitness : float    } ;;
```

La **population** sera donc de type `individu vect`.

Remarque Le dernier individu de la population servira uniquement à conserver, de génération en génération, le meilleur individu rencontré par l'algorithme génétique. Il ne sera pas modifié par des crossover ou des mutations. En revanche, après avoir effectué chacun de ces opérateurs sur le reste de la population, l'algorithme mettra à jour ce dernier individu si besoin est. D'autre part, ce meilleur individu sera pris en compte et «réinjecté» dans le reste de la population d'une génération sur l'autre, afin qu'une (ou plusieurs) copie(s) de cet individu subissent des modifications.

3.1.3 La fonction d'évaluation

La fonction d'évaluation(`evalue_ind`, cf 3.2) calculera la différence entre un temps maximum (`temps_max`, correspondant à un trajet passant par tous les aérodromes) et le temps de parcours du trajet. Afin de minimiser le temps de parcours, l'algorithme génétique devra rendre cette différence maximum.

3.1.4 Les opérateurs

Les opérateurs choisis ici sont la **mutation** et le **crossover à 2 points**. La sélection se fera selon le principe de la roulette biaisée.

3.1.5 Les individus non viables

Le problème que nous avons choisi d'étudier peut amener l'algorithme génétique à créer (par des mutations ou des crossovers) des individus qui représentent des trajets impossibles à réaliser pour le pilote, compte tenu de l'autonomie de son appareil. En effet l'avion a une autonomie limitée, et tout individu qui représente un trajet qui se solderait par un crash sera appelé par la suite **individu non viable**.

Nous avons choisi de détruire puis remplacer tout *individu non viable* que l'algorithme génétique rencontrera : pour cela nous utiliserons la fonction `individu_viable`, qui sera appelée après chaque modification du «brin d'ADN» susceptible de créer un individu non viable. Cette fonction, récursive, regarde si l'individu qu'on lui propose est "viable" (s'il est non viable, la fonction d'évaluation [cf 3.2] renvoie `-1`) : s'il ne l'est pas, elle en crée un nouveau (aléatoirement) et se rappelle elle-même, jusqu'à obtenir un individu viable.

Programme 1 (Obtention d'un individu viable)

```
let rec individu_viable ind mat_temps temps_rav temps_max =
  let long_trajet = vect_length ind.trajet in
  change_fitness ind mat_temps temps_rav temps_max ;
  if ind.fitness = -1.
  then
    begin
      let trajet_aleatoire = make_vect long_trajet 0 in
      for j = 0 to (long_trajet - 1)
      do
        let bit = random__int 2 in
        trajet_aleatoire.(j) <- bit
      done ;
      ind.trajet <- trajet_aleatoire ;
      individu_viable ind mat_temps temps_rav temps_max ;
    end
  else
   ind ;;
```

Accessoirement, cette fonction peut servir à générer les individus de la première génération, c'est-à-dire la *population initiale*.

3.1.6 La population initiale

Nous pouvons maintenant créer la population initiale, qui est générée aléatoirement. Pour cela, il nous suffit de créer des individus "vides" (c'est-à-dire dont le trajet ne contient que des `0`), et de leur appliquer la fonction précédente `individu_viable`, qui se chargera d'obtenir des individus viables. La fonction `initialise` suivante renvoie donc une population initiale faite uniquement d'individus viables.

Programme 2 (Création de la population initiale)

```
let initialise long_trajet nb_ind mat_temps temps_rav temps_max =
  let trajet_nul = make_vect long_trajet 0 in
  let ind_vide = {trajet = trajet_nul ; fitness =  0. ;
                      cfitness = 0.        ; rfitness = 0. } in
  let population = make_vect nb_ind ind_vide in
  for i = 0 to (nb_ind - 1 )
  do
    let ind_ok = individu_viable ind_vide mat_temps
                                          temps_rav temps_max in
    population.(i) <-  ind_ok
  done ;
  population ;;
```

3.2 Fonction d'évaluation

Il faut maintenant évaluer chaque individu. La fonction `evalue_ind` calcule le temps mis pour parcourir le trajet indiqué par l'individu. Pour cela, elle se réfère à la «matrice des temps» : maintenant que les aérodromes ont été numérotés, nous pouvons créer la «matrice des temps» $\mathcal{M}$, qui donne le temps de parcours $\mathcal{M}_{i,j}$ (pour un trajet direct) entre 2 aérodromes i et j, ou `-1` si l'avion n'a pas une autonomie suffisante pour rallier directement les aérodromes i et j. Nous indiquons donc d'abord l'emplacement des aérodromes sur la carte, par leurs coordonnées, en donnant les aérodromes dans l'ordre choisi précédemment :

```
let mat_ad =
 [|
  [|1.;8.|];[|3.;7.|];[|3.;10.|];[|4.;12.|];[|5.;7.|] ;
  [|5.;9.|];[|6.;5.|];[|6.;12.|];[|7.;7.|];[|7.;10.|] ;
  [|8.;9.|];[|9.;7.|];[|9.;12.|];[|10.;9.|];[|11.;13.|] ;
  [|12.;6.|];[|13.;8.|];[|15.;5.|];[|13.;11.|];[|15.;7.|] ;
  [|15.;13.|];[|16.;10.|];[|17.;8.|];[|19.;7.|];[|17.;12.|] ;
  [|18.;11.|];[|19.;13.|];[|20.;12.|];[|20.;10.|];[|21.;11.|] |] ;;
```

La fonction `cree_matrice`, comme son nom l'indique, crée la matrice des temps, à partir de la matrice des aérodromes "`a`", de l'autonomie "`aut`" et de la vitesse de l'avion "`vit`". Après avoir créé une matrice ne contenant que des `-1`, la fonction `cree_matrice` calcule tout simplement la distance entre les deux aérodromes : s'ils sont trop éloignés (pour une autonomie donnée), la fonction ne fait rien, sinon, elle remplace le `-1` par la durée du trajet entre les 2 aérodromes donnés.

Programme 3 (Création de la matrice des temps)

```
let cree_matrice a aut vit =
  let taille = vect_length a  in
  let m = make_matrix taille taille (-1.) in
  for i = 0 to ( taille - 1 )
  do
    for j = 0 to ( taille - 1 )
```

```
      do
        let x = ( a.(j).(0) -. a.(i).(0) )
            and y = ( a.(j).(1) -. a.(i).(1) ) in
        let dist = sqrt ( x *. x +. y *. y ) in
        if (dist <=. aut) || (aut =. -1.)
        then m.(i).(j) <- ( dist /. vit ) ;
     done
    done ;
    m ;;
```

Nous pouvons donc écrire maintenant la fonction `evalue_ind`, qui renvoie le temps mis pour parcourir le trajet donné (**en comptant le temps `temps_rav` nécessaire à chaque ravitaillement**), ou `-1` si le trajet se solde par un crash...

Programme 4 (Evaluation d'un individu)

```
let evalue_ind ind mat_temps temps_rav temps_max =
  let trajet = ind.trajet in
  let longueur = vect_length trajet in
  let trajet_a_evaluer = make_vect (longueur + 2) 0 in
  trajet_a_evaluer.(0) <- 1 ;
  trajet_a_evaluer.(longueur + 1) <- 1 ;
  for i=1 to longueur
  do
    trajet_a_evaluer.(i) <- trajet.(i-1)
  done;
  let dernier_posé = ref (-1) and temps = ref 0.
   and compteur = ref 0 in
  let i = ref 0 in
  while ( !i <= (longueur + 1) ) && ( !temps <> -1. )
  do
    if trajet_a_evaluer.(!i) = 1
    then
      begin
        incr compteur ;
        if !compteur > 1
        then
          begin
            let temps_int = mat_temps.(!dernier_posé).(!i) in
            if temps_int = -1.
            then   temps := -1.
            else    temps := !temps +. temps_int
          end ;
      end;
    if trajet_a_evaluer.(!i) = 1  then dernier_posé := !i ;
    incr i ;
  done ;
  let fitness = ref 0. and
     temps_rav_total =
         ((float_of_int (!compteur - 2)) *. temps_rav) in
  if !temps = -1.
  then fitness := -1.
  else fitness :=
         (temps_max -. ( !temps +. temps_rav_total )) ;
```

```
!fitness ;;
```

Deux fonctions annexes nous seront utiles :

– `change_fitness` qui modifie la valeur de la fitness de l'individu, en la recalculant (après toute opération qui modifie le trajet de l'individu, il faut recalculer le temps de parcours) :

Programme 5 (Réactualisation de la fitness d'un individu)

```
let change_fitness ind mat_temps temps_rav temps_max =
  let fitness = (evalue_ind ind mat_temps temps_rav temps_max)
  in ind.fitness <- fitness ;;
```

– `evalue_pop` qui met à jour toutes les fitness des individus d'une population :

Programme 6 (Evaluation de toute la population)

```
let evalue_pop population mat_temps temps_rav temps_max =
   let nb_ind = vect_length population in
   for i = 0 to (nb_ind - 1)
   do
    change_fitness population.(i) mat_temps
                         temps_rav temps_max
   done ;;
```

3.3 Mutation

Ici nous appliquerons la mutation uniforme : la probabilité de mutation s'entend comme probabilité sur *tous* les gènes. Si par exemple on choisit une probabilité de 0.3, cela signifie qu'environ 30 % des *gènes* vont être mutés, et non que 30 % des individus vont être mutés une fois. Cela permet ainsi à un individu de muter plusieurs fois dans la même génération. Par contre, une fois que toutes les mutations sur un individu ont été faites, si on aboutit à un individu non viable, nous avons ici choisi de conserver l'individu tel qu'il était avant la dernière mutation qui l'a rendu non viable. La fonction `mutation` suivante modifie une population *en place* (elle ne renvoie pas de résultat), en faisant muter aléatoirement les gènes de ses individus :

Programme 7 (Opérateur mutation)

```
let mutation population proba mat_temps temps_rav temps_max =
   let nb_ind = vect_length population in
   let long_trajet = vect_length population.(0).trajet in
   for i = 0 to (nb_ind - 2)
   do
      let individu_test = {
                  trajet = copy_vect population.(i).trajet ;
                  fitness = 0. ; cfitness = 0. ;
                  rfitness = 0. } in
       change_fitness individu_test mat_temps
                  temps_rav temps_max ;
```

```
    let individu_viable = {
                trajet = copy_vect population.(i).trajet ;
                fitness = population.(i).fitness ;
                cfitness = 0. ; rfitness = 0. } in
    for j = 0 to long_trajet - 1
    do
      let x = (random__float 1000. ) /. 1000. in
      if x <. proba
      then
        begin
          individu_test.trajet.(j) <- (1 - individu_test.trajet.(j)) ;
          change_fitness individu_test
                mat_temps temps_rav temps_max ;
          if individu_test.fitness <> -1.
          then
            individu_viable.trajet <- copy_vect individu_test.trajet ;
        end;
    done ;
    population.(i).trajet <- copy_vect individu_viable.trajet ;
    change_fitness population.(i) mat_temps temps_rav temps_max ;
done ;;
```

3.4 Crossover

Le crossover choisi est le crossover à deux points, permettant de modifier le début et la fin des individus en même temps. Pour réaliser un crossover, l'algorithme génétique utilisera deux fonctions : `select_crossover` et `effectue_crossover`.

La première, `select_crossover`, choisira les deux "parents" auxquels on effectuera le crossover, en fonction de la probabilité de crossover, qui correspond ici à la probabilité qu'a chaque individu d'être sélectionné pour subir un crossover.

Programme 8 (Sélection des individus parents)

```
let select_crossover population proba  mat_temps temps_rav temps_max  =
  let nb_ind = vect_length population in
  let parent_un = ref 0 in
  let parents = ref 0 in
  for i = 0 to (nb_ind - 2)
  do
    let x = (random__float 1000.) /. 1000. in
    if x <. proba
    then
      begin
        incr parents ;
        if (!parents mod 2) = 0
        then
           effectue_crossover population !parent_un i
                             mat_temps temps_rav temps_max
        else
          parent_un := i ;
      end;
  done ;;
```

Cette fonction fait appel à la deuxième, `effectue_crossover`, qui réalise effectivement le crossover, en choisissant aléatoirement deux points d'application. L'algorithme décompose les brins d'ADN des deux parents aux endroits correspondants, et les recompose pour obtenir les deux «enfants». Si l'un des deux enfants nés du crossover est non viable, l'algorithme réessaie avec deux autres points (aléatoires) ; au bout de 10 essais (pour éviter les cas où les crossover sont pratiquement irréalisables, du fait de la position des différents aérodromes ; cf plus loin, paragraphe 4.2.1), l'algorithme génétique arrête ses essais et conserve les «parents» sans les modifier.

Programme 9 (Opérateur crossover)

```
let effectue_crossover population
      parent1 parent2 mat_temps temps_rav temps_max =
  let long_trajet = vect_length population.(0).trajet in
  let point1 = ref 0 and point2 = ref 0 in
  if long_trajet > 1
  then
    begin
      let fils_test1 = {trajet = copy_vect population.(parent1).trajet ;
                          fitness = -1. ; cfitness = 0. ; rfitness = 0. }
      and fils_test2 =  {trajet = copy_vect population.(parent2).trajet ;
                          fitness = -1. ; cfitness = 0. ; rfitness = 0. } in
      let essais = ref 0 in
      while (!essais <= 10) &&
                ((fils_test1.fitness = -1. ) || (fils_test2.fitness = -1.))
      do
         fils_test1.trajet <- copy_vect population.(parent1).trajet ;
         fils_test2.trajet <- copy_vect population.(parent2).trajet ;
         if long_trajet = 2
         then  point1 := 0
         else
           begin
              point1 := random__int long_trajet ;
              point2 := !point1 + (random__int (long_trajet - !point1)) ;
           end ;

         for i = 0 to !point1
         do
           let temp = fils_test1.trajet.(i) in
           fils_test1.trajet.(i) <- fils_test2.trajet.(i) ;
           fils_test2.trajet.(i) <- temp ;
         done ;
         for i = !point2 to (long_trajet - 1)
         do
           let temp = fils_test1.trajet.(i) in
           fils_test1.trajet.(i) <- fils_test2.trajet.(i) ;
           fils_test2.trajet.(i) <- temp ;
         done ;
         change_fitness fils_test1 mat_temps temps_rav temps_max ;
         change_fitness fils_test2 mat_temps temps_rav temps_max ;

         incr essais ;
        done ;
```

```
            if !essais <= 10
            then
              begin
                 population.(parent1) <- fils_test1 ;
                 population.(parent2) <- fils_test2 ;
              end  ;
    end ;;
```

3.5 Sélection

3.5.1 La roulette biaisée

La séléction des individus qui survivront se fera sur le principe de la roulette biaisée : pour chaque individu, on calcule sa fitness relative (`rfitness`), c'est-à-dire sa fitness rapportée à la somme des fitness (calculée précédemment). Ensuite, l'algorithme génére un nombre aléatoire et l'individu possédant la «marche d'escalier » correspondante (voir la théorie, paragraphe 2.2.3) est recopié dans la nouvelle génération.

Programme 10 (Roulette biaisée)

```
let selection population =
  let nb_ind = vect_length population in
  let nouvelle_population = copy_vect population in
  let total_fitness = ref 0. in
  for i = 0 to (nb_ind - 1)
  do
     total_fitness := !total_fitness +. population.(i).fitness
  done ;
  for i = 0 to (nb_ind - 1)
  do
     population.(i).rfitness <- population.(i).fitness /. !total_fitness
  done ;
  population.(0).cfitness <- population.(0).rfitness;
  for i = 1 to (nb_ind - 1)
  do
     population.(i).cfitness <- population.(i-1).cfitness +. population.(i).rfitness
  done ;

  for i = 0 to (nb_ind - 2)
  do
    let p = (random__float 1000.) /. 1000. in
    if p <. population.(0).cfitness
    then  nouvelle_population.(i) <- population.(0)
    else   begin
              for j = 0 to (nb_ind - 2)
              do
                 if (p >=. population.(j).cfitness) && (p <. population.(j+1).cfitness)
                 then nouvelle_population.(i) <- population.(j+1) ;
              done
            end
  done ;

  for i = 0 to (nb_ind - 2)
```

```
    do  population.(i) <- nouvelle_population.(i)
    done ;;
```

3.5.2 La conservation du meilleur individu

Notre but est d'obtenir la meilleure solution possible au problème posé. Comme les opérateurs modifient constamment les individus de la population, il convient de conserver le meilleur individu, **toutes générations confondues**. La fonction `garde_le_meilleur` le conservera en lieu et place du dernier individu de la population : les opérateurs pouvant modifier les individus (crossover, mutation) ne l'affecteront pas. Pour conserver le meilleur individu à la dernière position, il nous faut pouvoir remplacer un individu par un autre au sein de la population : la fonction `remplace` réalise ceci.

Programme 11 (Remplacement d'un individu par un autre)

```
let remplace population ind_a_remplacer ind_qui_remplace =
   let trajet_de_remplacement = (population.(ind_qui_remplace)).trajet
     and fitness_de_remplacement = (population.(ind_qui_remplace)).fitness in
   population.(ind_a_remplacer) <-  {trajet = trajet_de_remplacement ;
                                      fitness = fitness_de_remplacement ;
                                      cfitness = 0. ; rfitness = 0. } ;;
```

Nous pouvons maintenant écrire la fonction `garde_le_meilleur` :

Programme 12 (Conservation du meilleur individu)

```
let garde_le_meilleur population =
   let nb_ind = vect_length population in
   let le_meilleur = ref 0 and best_fitness = ref 0. in
   for i = 0 to (nb_ind - 1)
   do
     if (population.(i)).fitness >=. !best_fitness
     then
       begin
         le_meilleur := i ;
         best_fitness := (population.(i)).fitness ;
       end;
   done;
   remplace population (nb_ind - 1) !le_meilleur ;;
```

3.5.3 La fonction élitiste

Le meilleur individu est souvent celui qui est le plus proche de la solution optimale. L'ayant conservé en dernière position, nous pouvons comparer sa fitness à celles des individus de la nouvelle génération (la nouvelle génération étant les individus de 1 à $n-1$). Si l'algorithme n'a pas trouvé mieux dans la nouvelle génération, la fonction `elitiste` va «réinjecter» le meilleur individu de la génération précédente dans la population afin que l'algorithme génétique puisse essayer des mutations et des crossover sur cet individu.

Programme 13 (Fonction élitiste)

```
let elitiste population =
   let nb_ind = vect_length population in
   let le_meilleur = ref 0 and le_pire = ref 0 in
   for i = 0 to nb_ind - 2
   do
     if population.(i).fitness >. population.(!le_meilleur).fitness
     then le_meilleur := i ;
     if population.(i).fitness <. population.(!le_pire).fitness
     then le_pire := i ;
   done ;
   if population.(!le_meilleur).fitness <. population.(nb_ind - 1).fitness
   then remplace population (!le_pire) (nb_ind - 1)
   else remplace population (nb_ind - 1) (!le_meilleur) ;;
```

3.6 Fonction globale

3.6.1 Fonction principale

Il reste à écrire la fonction principale, `algo_genetique`, appelant toutes ces sous-fonctions. La liste des arguments nécessaires à l'exécution de l'algorithme est donc la suivante :

- `mat_ad` : La matrice des coordonnées des aérodromes
- `aut` : L'autonomie de l'avion
- `vit` : La vitesse de l'avion
- `temps_rav` : Le temps nécessaire à chaque ravitaillement
- `nb_ind` : Le nombre d'individus dans la population
- `proba_cross` : La probabilité de crossover
- `proba_mut` : La probabilité de mutation
- `nb_gen` : Le nombre de générations au bout duquel l'algorithme doit s'arrêter
- `f_arret` : Afin d'éviter à l'algorithme de continuer à travailler bien qu'il ait trouvé la solution optimale (dans le cas où celle-ci est connue...), on peut lui indiquer une fitness maximale : s'il l'atteint, il s'arrête. Si bien sûr la solution optimale n'est pas connue à l'avance, on donne une valeur très grande (impossible à atteindre) à cet argument.

Programme 14 (Fonction principale)

```
let algo_genetique mat_ad aut vit temps_rav nb_ind proba_cross proba_mut nb_gen f_arret =
   let long_trajet = ( (vect_length mat_ad) - 2 ) in
   let mat_temps = ( cree_matrice mat_ad (-1.) vit ) in
   let plus_long_trajet = make_vect long_trajet 1 in
   let ind_long = { trajet = plus_long_trajet ; fitness = 0. ;
                          rfitness = 0. ; cfitness = 0. } in
   let temps_max = minus_float (evalue_ind ind_long mat_temps temps_rav 0.) in
   let mat_temps = ( cree_matrice mat_ad aut vit ) in
   let population = ( initialise long_trajet nb_ind mat_temps temps_rav temps_max ) in
      imprime [| String "**-Population créée avec "; Int nb_ind ;
                    String "individus-**" |] ecran ;
   evalue_pop population mat_temps temps_rav temps_max ;
   garde_le_meilleur population ;
   let gen = ref 0 in
   while (!gen < nb_gen)  && (population.(nb_ind - 1).fitness <. f_arret )
   do
```

```
    incr gen ;
    selection population ;
        garde_le_meilleur population ;
    select_crossover population proba_cross mat_temps temps_rav temps_max ;
        garde_le_meilleur population ;
    mutation population proba_mut mat_temps temps_rav temps_max ;
       garde_le_meilleur population ;
    evalue_pop population mat_temps temps_rav temps_max ;
        garde_le_meilleur population ;
    elitiste population ;
        garde_le_meilleur population ;
  imprime [|String "Gén :" ; Int !gen ;
             String "Meilleur individu :" ; Trajet population.(nb_ind - 1).trajet ;
             String "Fitness : ";
             Float  population.(nb_ind - 1).fitness ;
             Newline  |] ecran ;
  done ;
  garde_le_meilleur population ;
  imprime [|String "Gén. :" ; Int !gen ;
             String "Meilleur :" ; Trajet population.(nb_ind - 1).trajet ;
             Newline; String "Ce qui correspond à un temps de parcours de : ";
             Float (temps_max -. population.(nb_ind - 1).fitness) ;
             Newline  |] true ;
  (!gen,population.(nb_ind - 1).fitness,population.(nb_ind - 1).trajet) ;;
```

La fonction principale fait appel également à la fonction `imprime`, une fonction d'affichage des données sous forme simplifiée.

3.6.2 Fonctions d'affichage des données

Les données sont de plusieurs types, et les fonctions à utiliser pour les afficher varient selon leur type. Le type `donnée` va rassembler tous ces types :

```
type donnée =
    |Newline
    |Int of int
    |Float of float
    |String of string
    |Int_vect of int vect
    |Trajet of int vect
    |Float_vect of float vect ;;
```

Voici deux fonctions permettant d'afficher de façon plus lisible les vecteurs d'entiers, de flottants.

Programme 15 (Affichage d'un vecteur d'entier)

```
let print_int_vect a =
  let long = vect_length a in  print_string "(" ;  print_int  a.(0);
  for i = 1 to (long - 1)
  do print_string "," ; print_int  a.(i);   done ;
  print_string ")" ;;
```

Programme 16 (Affichage d'un vecteur de flottants)

```
let print_float_vect a =
  let long = vect_length a in print_string "(" ; print_float  a.(0);
  for i = 1 to (long - 1) do print_string "," ; print_float  a.(i); done ;
  print_string ")" ;;
```

Une fonction affichant les trajets (vecteurs de 0 et de 1) sous la forme de la liste des aérodromes sur lesquels le pilote va se poser. Par exemple le trajet [—0,1,0,0,0,0,0,0,1,0,0,1,0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,0—] sera affiché sous la forme : (3,10,13,19,25,).

Programme 17 (Affichage d'un trajet)

```
let print_trajet a =
  let long = vect_length a in print_string "(" ;
  for i = 0 to (long - 1)
  do  if a.(i) = 1  then begin print_int (i+2) ; print_string "," ; end ; done ;
  print_string ")" ;;
```

Ces fonctions écrites, la fonction `imprime_donnee` va, en fonction du type de donnée, appeler la fonction d'affichage correspondante.

Programme 18 (Affichage de n'importe quel type de données)

```
let imprime_donnée d = match d with
    |Newline -> print_newline ()
    |Int a -> print_int a
    |Float a -> print_float a
    |String a -> print_string a
    |Int_vect a -> print_int_vect a
    |Trajet a -> print_trajet a
    |Float_vect a -> print_float_vect a  ;;
```

Enfin voici la fonction `imprime` (appelée par la fonction principale `algo_genetique`), qui affiche tous les éléments d'un «vecteur de données» (le paramètre `ecran` permet de n'afficher que certaines choses, suivant les tests qu'on réalise) :

Programme 19 (Affichage d'un vecteur de données)

```
let imprime v ecran =
   if ecran
   then
     begin
      for i = 0 to (vect_length v) - 1
      do print_string " ";  imprime_donnée v.(i) ; print_string " ";
      done;
     end ;;
```

4 Résultats et analyses

Maintenant que le problème a été entièrement codé en CAML, étudions les résultats des différents tests réalisés grâce à notre algorithme génétique.

La première surprise a été de constater que **l'algorithme génétique avait trouvé mieux que nous !** En effet, dès les premiers tests, effectués avec les paramètres suivants :

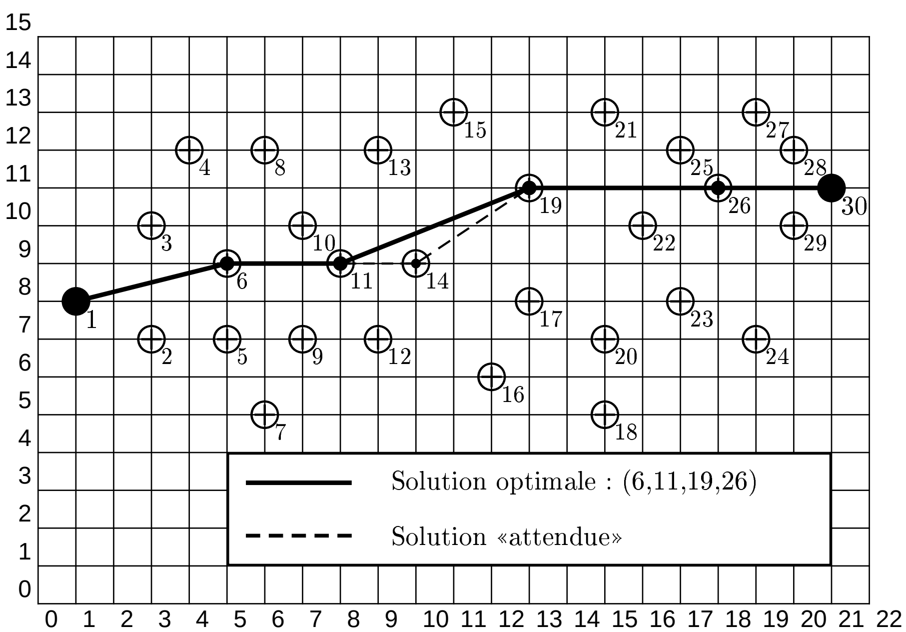
- Carte des aérodromes : matrice `mat_ad` (voir page 14).
- Autonomie : 600 km, c'est-à-dire `6.` selon l'unité choisie (100 km).
- Vitesse : 100 km.h^{-1}, c'est-à-dire `1.` selon l'unité choisie.
- Durée d'un ravitaillement : `0.75` heure.
- Nombre d'individus dans la population : `20`.
- Probabilité de crossover : `0.3`.
- Probabilité de mutation : `0.2`.
- Nombre maximum de générations : `5 000`.
- Fitness d'arrêt : `10 000` (un nombre arbitraire, très grand, pour que l'algorithme continue de chercher pendant 5000 générations [voir `f_arret`, page 21).

[ce qui donne en CAML : `algo_genetique mat_ad 6. 1. 0.75 20 0.3 0.2 5000 10000`]

l'algorithme nous a renvoyé une solution à laquelle nous n'avions pas pensé : **(6,11,19,26)** !

Celle-ci est représentée figure 7.

FIG. 7 – SOLUTION OBTENUE



L'algorithme a «trouvé» une amélioration au chemin auquel nous nous attendions : en effet, après s'être posé en **6**, il ne fallait pas chercher à utiliser toute son autonomie (de 6 cases) pour se poser le plus loin possible, en **14**, mais se poser en **11**, pour pouvoir ensuite raccourcir le temps de vol !

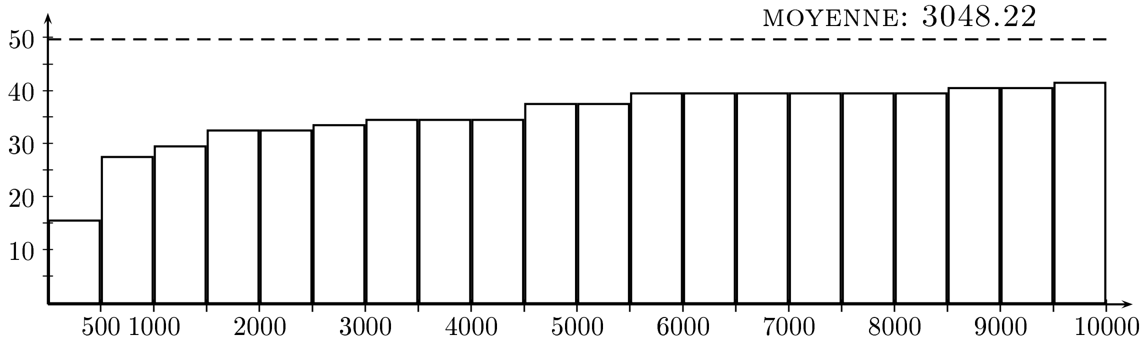
Après vérification avec d'autres méthodes (de parcours de graphes), il s'est avéré que cette solution était bien la solution optimale...

C'est donc finalement à partir de cette solution optimale que nous avons pu étudier nos tests, en variant les paramètres (notamment les probabilités de crossover et de mutation) : nous avons à chaque fois regardé au bout de combien de générations l'algorithme arrivait à trouver cette solution optimale.

Nous avons réalisé des séries de 50 tests, chacun limité à 10 000 générations (nous avons poussé le calcul un peu plus loin que lors des premiers tests). **Nous avons reporté ces résultats sous formes d'histogrammes présentant en ordonnée le nombre de fois où l'algorithme a trouvé la solution optimale (6,11,19,26) en moins de X générations, où X est l'abscisse** : on évalue ainsi la rapidité de l'algorithme à se rapprocher de cette solution. On donne également sur ces graphiques **la moyenne** sur les 50 tests du nombre de générations au bout duquel la solution optimale a été trouvée.

Ce qui donne, par exemple, pour le test «témoin», la figure 8.

Fig. 8 – Paramètres : Crossover 0.5, Mutation 0.1 (témoin)



Ainsi, sur cette série de 50 tests, on voit que l'algorithme a trouvé la solution optimale en moins de 1000 générations plus d'une fois sur deux (28 fois sur 50 pour être exact).

Le temps de calcul, sur un ordinateur «actuel» (Pentium III) est d'environ 1 minute pour 1000 générations, avec une population de 20 individus.

4.1 Influence des paramètres

Nous allons dans cette section montrer le caractère indispensable des opérateurs crossover et mutation, et chercher les probabilités de crossover et de mutation pour lesquelles l'algorithme est le plus «rapide».

L'opérateur Mutation

Remarquons tout d'abord que l'opérateur Mutation est indispensable au bon fonctionnement d'un algorithme génétique. En effet si l'on exécute l'algorithme avec une probabilité de mutation de `0.`, on se rend compte qu'il n'arrive jamais à trouver une solution correcte au problème en moins de 10000 générations. Ceci est du au fait qu'aucune **exploration** du domaine des solutions n'est possible, puisque l'opérateur mutation n'entre plus en jeu.

De même, avec une probabilité de `1.`, aucune **exploitation** des solutions potentielles n'est possible, puisqu'à chaque génération, tous les individus voient tous leurs gènes muter : la population ne conserve plus la trace de ses meilleurs individus.

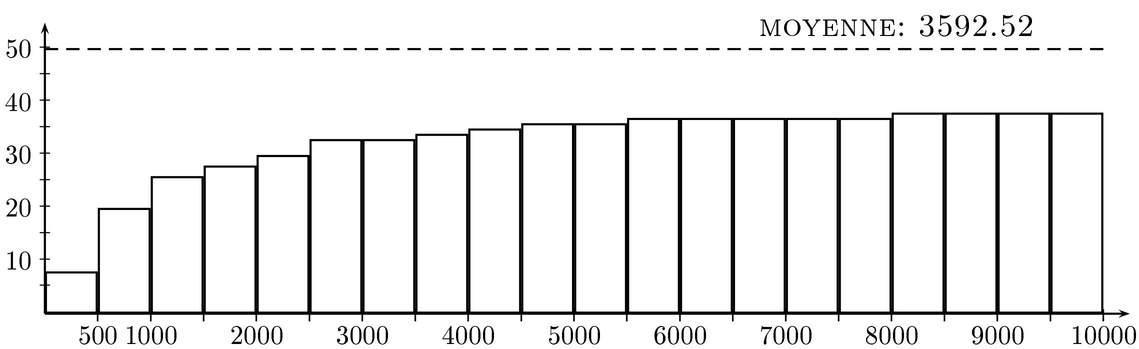
Nous obtenons ainsi des individus tels les quelques exemples répertoriés à la figure 9. Ils sont à comparer à l'individu idéal que nous avons déjà signalé : (6,11,19,26).

Fig. 9 – Cas Pathologiques

Expérience 1 Crossover 0.5 , Mutation 0.0	Expérience 2 Crossover 0.5 , Mutation 1.0
(3,11,16,17,19,21,22,26)	(3,8,11,16,23)
(3,5,10,14,16,17,23,24,29)	(7,12,16,18,23)
(5,6,9,14,17,23,28)	(3,8,14,17,25,28)
(2,3,11,12,17,22)	(7,12,20,24,27,28)
(6,9,16,18,20,23)	(6,13,19,21,22,27)
(2,5,7,12,14,16,22,25)	(2,12,16,18,23)
(3,10,11,12,16,18,22,28)	(6,10,15,21,22,26,29)
(2,4,6,13,14,17,25,26)	(7,9,16,23)

Regardons maintenant de plus près quelles seraient les valeurs intéressantes à donner à la probabilité de mutation pour que l'algorithme s'en trouve amélioré. Pour cela observons les graphiques des figures 10, 11, et 12 présentant les résultats de différentes expériences réalisées pour des valeurs diverses de probabilité de mutation (tout en conservant la probabilité de crossover initiale de 0.5.)

Fig. 10 – Paramètres : crossover 0.5, mutation 0.2



En observant les résultats donnés par ces différents diagrammes, nous remarquons que plus la probabilité de mutation augmente et moins l'algorithme s'approche de la solution optimale rapidement, autrement dit, moins il est efficace. Si nous voulons donc améliorer les performances d'un tel algorithme, **il va être préférable de choisir une probabilité de mutation assez faible, de l'ordre de 0.1.**.

Une trop grande probabilité de mutation privilégie en effet l'exploration par rapport à l'exploitation, et l'algorithme génétique peine à trouver rapidement de bonnes solutions, puisqu'il les modifie beaucoup à chaque génération.

Fig. 11 – Paramètres : crossover 0.5, mutation 0.4

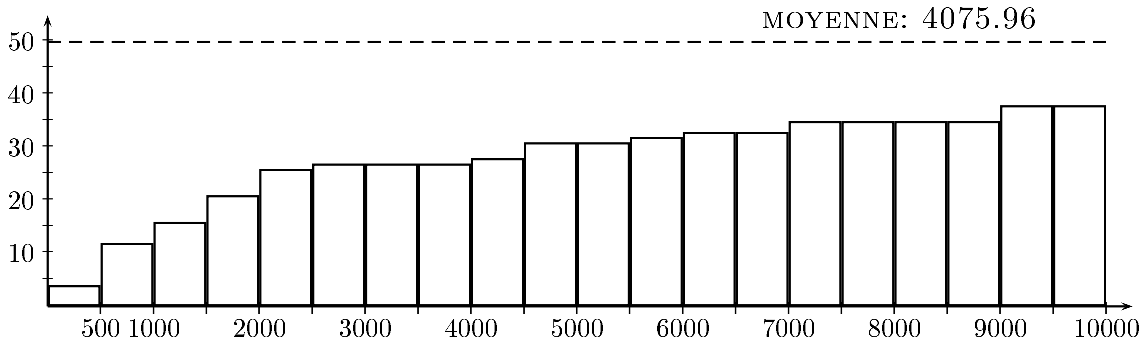
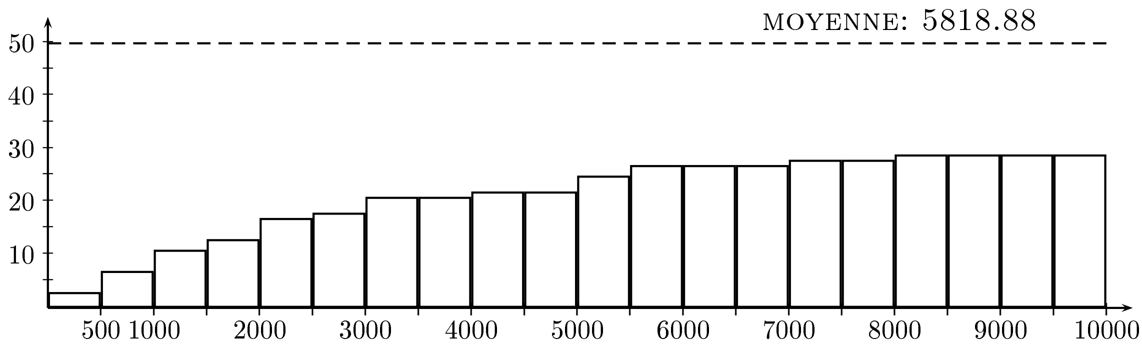


Fig. 12 – Paramètres : crossover 0.5, mutation 0.9



Remarque : Le temps de calcul est également un paramètre qu'il convient de prendre en compte, surtout lorsque l'on parle de mutation. On se rend compte en effet que lorsque l'on augmente la probabilité de mutation le temps de calcul grandit de manière impressionnante. On passe ainsi de trois quart d'heure environ (pour les 50 tests) pour une probabilité de 0.1 à plus de dix heures et demi pour une probabilité de 1.0. ! Ceci s'explique assez logiquement : plus l'on augmente la probabilité de mutation et plus le nombre de gènes changés dans le patrimoine d'un individu est grand. Ainsi l'on n'est pas du tout sûr d'obtenir un individu viable à l'arrivée. Dans le cas défavorable, l'algorithme génère alors d'autres individus, grâce à la fonction `individu_viable` (cf page 13) jusqu'à en trouver un nouveau viable, ce qui augmente largement le temps de calcul.

L'opérateur Crossover

Là encore, on s'aperçoit que l'algorithme génétique a **besoin** de cet opérateur. Il est inutile de l'illustrer à nouveau ici, mais en effectuant le test *probabilité de crossover 0.0, probabilité de mutation 0.1* on se rend compte que l'on obtient le même genre d'individus que lors des tests *probabilité de crossover 0.5, probabilité de mutation 0.0* ou *probabilité de crossover 0.5, probabilité de mutation 1.0* dont nous avons parlé dans le paragraphe précédent.

En effet si l'opérateur crossover n'est plus utilisé, l'**exploitation** des solutions obtenues est réduite à néant. L'algorithme obtient alors des individus, grâce à la mutation, **mais ceux-ci ne**

sont pas recombinés : sans la décomposition / recomposition des individus, l'algorithme «tourne en rond» et ne peut se rapprocher de la solution optimale que par des «coups de chance».

Passons donc directement à la détermination de la meilleure probabilité de crossover en s'aidant des graphiques des figures 13, 14, 15, 16 et 17.

Fig. 13 – Paramètres : crossover 0.1, mutation 0.1

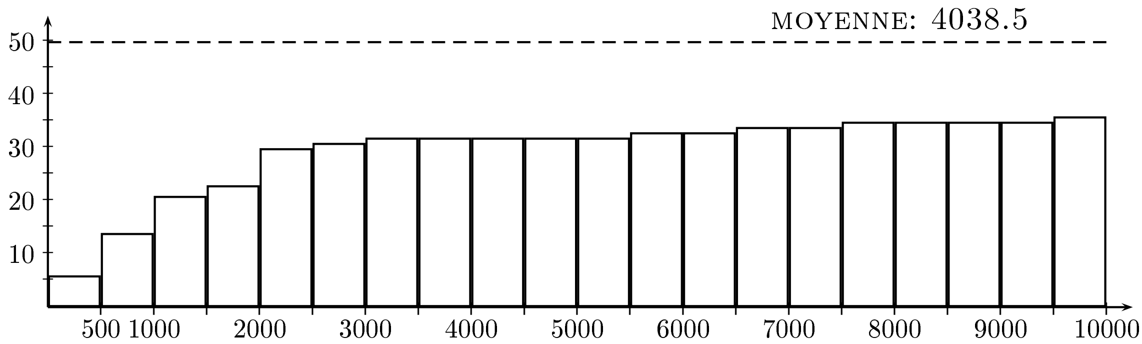


Fig. 14 – Paramètres : crossover 0.3, mutation 0.1

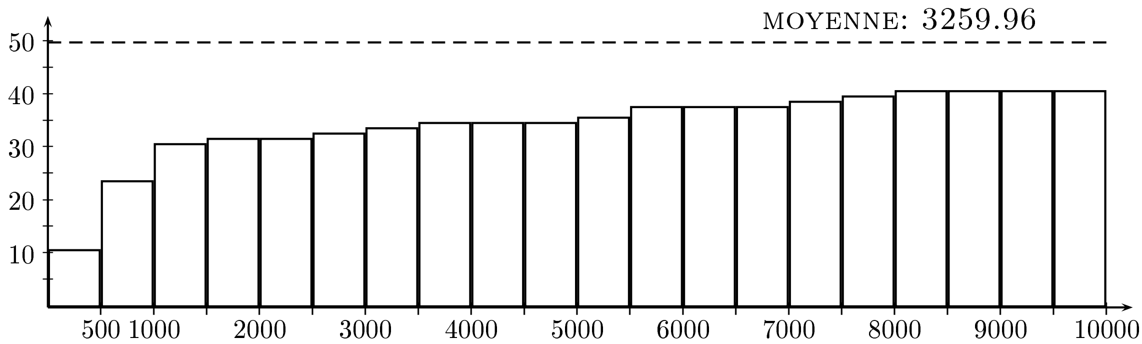


Fig. 15 – Paramètres : Crossover 0.5, Mutation 0.1 (témoin)

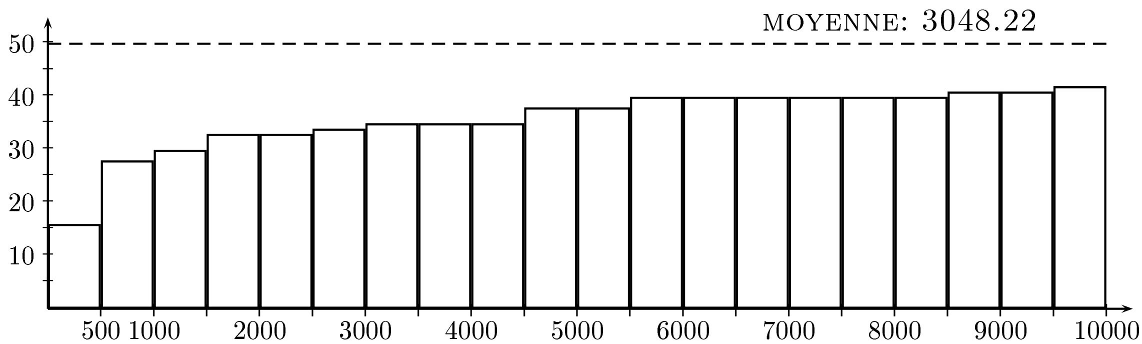


Fig. 16 – Paramètres : Crossover 0.8, Mutation 0.1

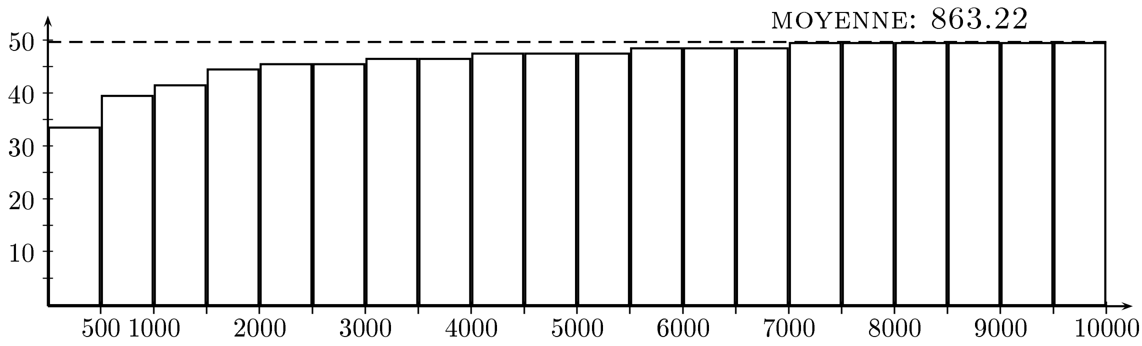
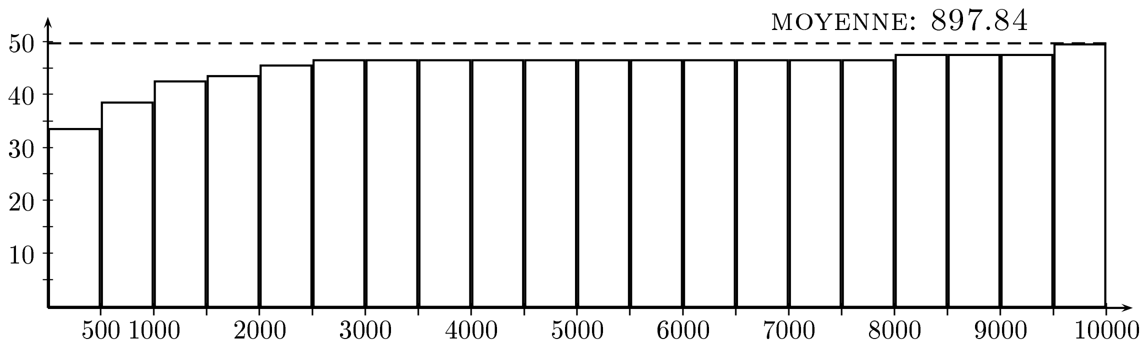


Fig. 17 – Paramètres : Crossover 1.0, Mutation 0.1



Les écarts importants dans les résultats observés laissent penser que le choix de la probabilité de crossover est assez déterminant. Si l'on se réfère à la moyenne comme indicateur de la «vitesse» de l'algorithme, **c'est pour une probabilité de crossover de `0.8` que l'algorithme est le plus performant**.

Nous avons donc réussi à dégager deux valeurs des paramètres pour lesquelles l'algorithme semble fonctionner le mieux :

Probabilité de crossover : `0.8` ; Probabilité de mutation `0.1`.

Mais nous avons modifié ces paramètres séparément. Nous avons donc réalisé une série de tests en faisant varier les deux paramètres *probabilité de mutation et de crossover* **ensemble**.

Variation des deux paramètres en même temps

Les figures 18, 19, 20, 21, 22 et 23 présentent les résultats des tests réalisés pour des valeurs des paramètres voisines des deux valeurs que nous avons mises en évidence dans les deux paragraphes précédents :

Fig. 18 – Paramètres : Crossover 0.7, Mutation 0.4

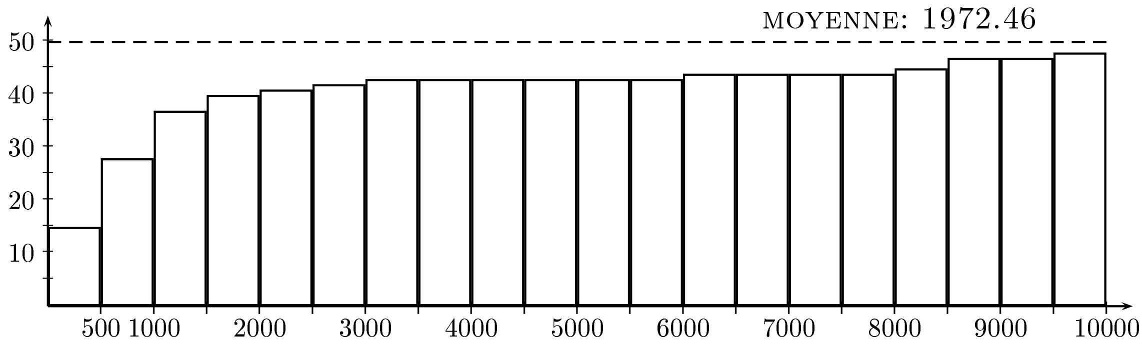


Fig. 19 – Paramètres : Crossover 0.7, Mutation 0.1

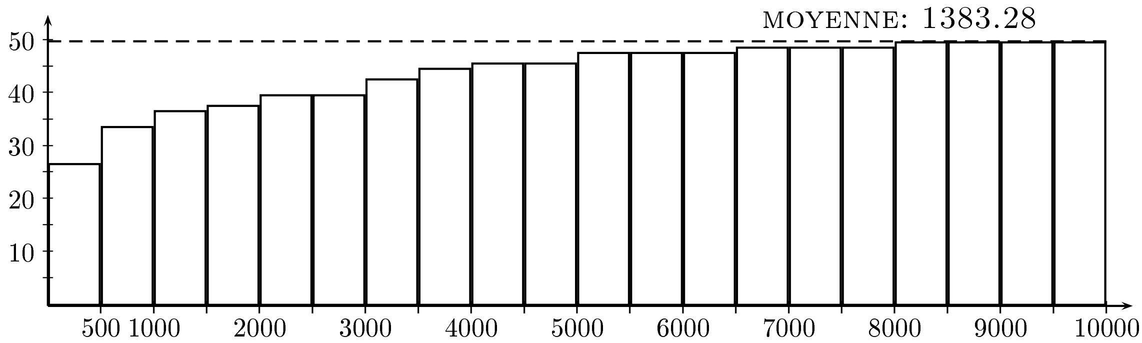


Fig. 20 – Paramètres : Crossover 0.9, Mutation 0.4

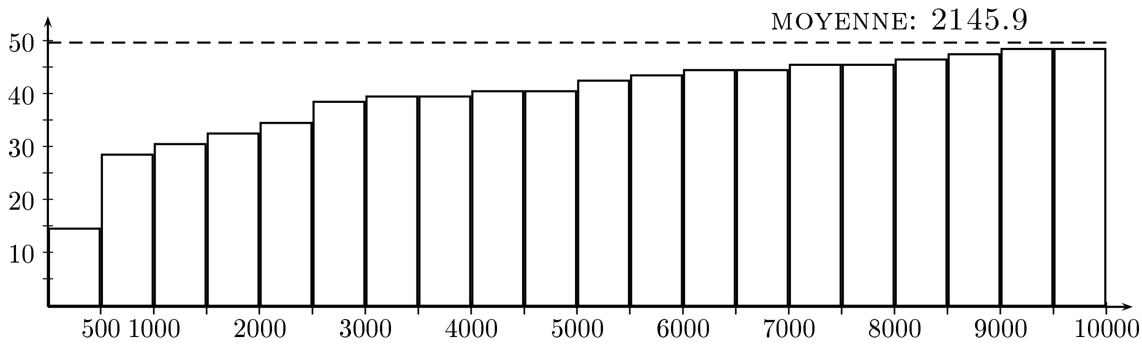


Fig. 21 – Paramètres : Crossover 0.9, Mutation 0.1

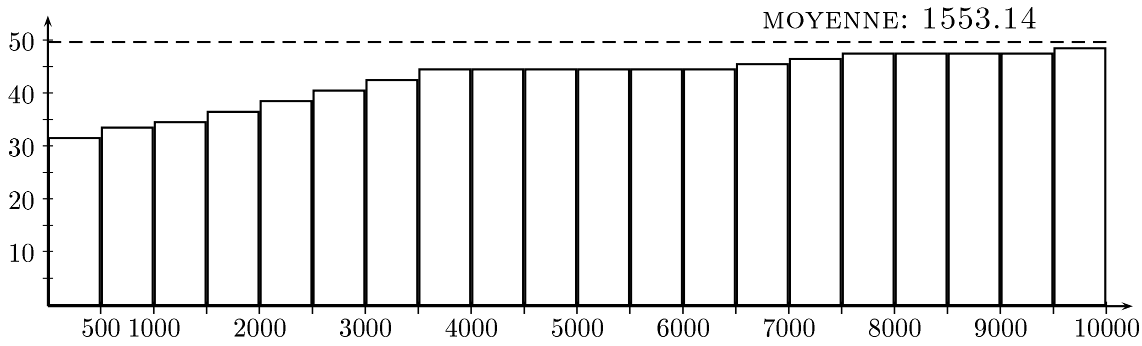


Fig. 22 – Paramètres : Crossover 1.0, Mutation 0.4

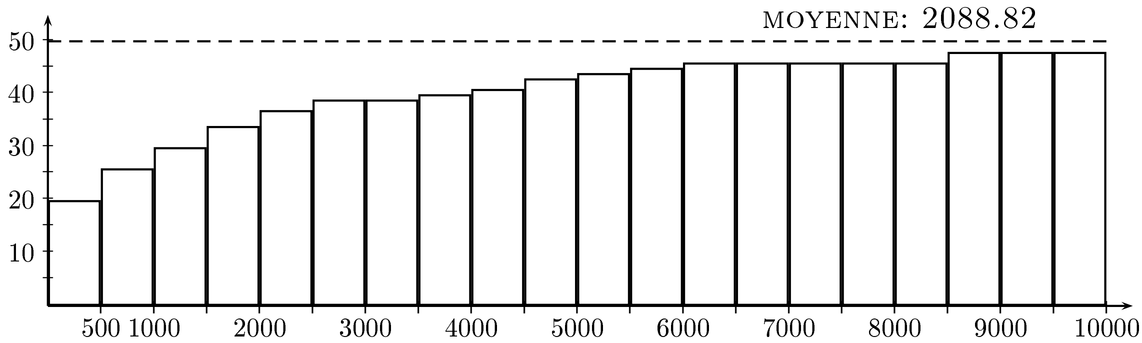
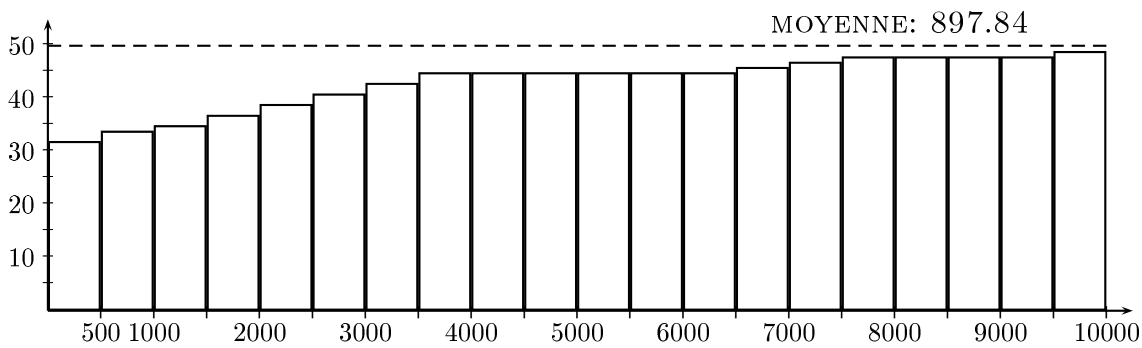


Fig. 23 – Paramètres : Crossover 1.0, Mutation 0.1



Si l'on regarde la moyenne (et l'allure générale) des diagrammes 18, 19, 20, 21, 22 et 23, les résultats que nous avions obtenus se confirment. Les deux paramètres, pour cet exemple, donnent les meilleurs résultats (en terme de «vitesse» de l'algorithme) avec les valeurs suivantes :

Probabilité de Crossover 0.8, Probabilité de mutation 0.1

4.2 Influence de la spécificité de l'exemple

Nous avons «affiné» notre algorithme génétique en cherchant les valeurs optimales des probabilités de crossover et de mutation. Mais nous avons ensuite cherché à savoir si notre algorithme était apte à résoudre d'autres problèmes, notamment si l'on change la carte des aérodromes ou les caractéristiques de l'appareil (vitesse, autonomie, durée du ravitaillement).

En effet notre algorithme fonctionne très bien pour résoudre le problème posé, mais c'est parce que celui-ci est simple : nous connaissions (à peu de chose près !) la solution avant que l'algorithme ne la trouve. Nous avons voulu savoir si dans d'autres conditions, notre algorithme était toujours efficace.

Les premiers essais furent menés en changeant la vitesse de l'appareil : cela ne changeait pratiquement rien au résultat.

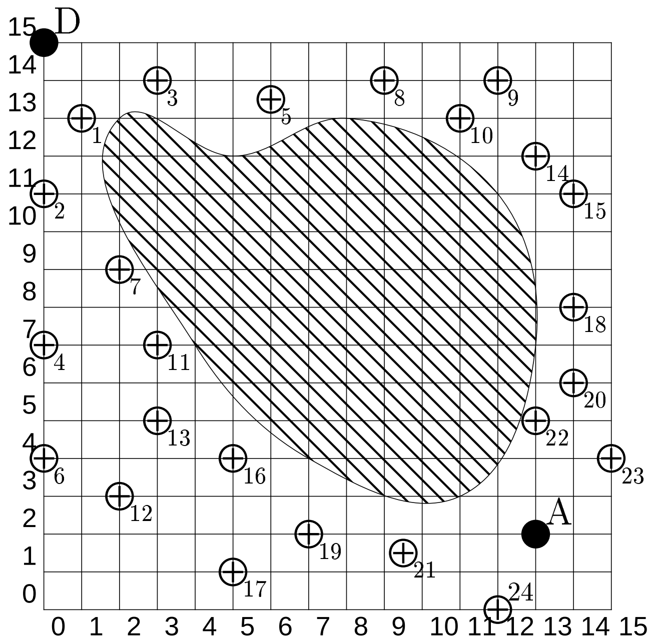
Puis nous avons modifié l'autonomie : en l'augmentant, l'algorithme génétique a bien «réagi» à ces nouvelles données, et le parcours qu'il conseillait contenait de moins en moins de ravitaillements. A la limite, **en prenant une autonomie «infinie»** (en pratique, plus grande que la plus grande distance entre 2 aérodromes, par exemple 5000 km d'autonomie), **l'algorithme génétique nous a bien renvoyé** (et très rapidement) un parcours ne contenant que des «**0**», codant donc **un trajet direct entre le départ et l'arrivée** (ce qui est bien sûr la meilleur solution !).

En modifiant beaucoup la carte, nous sommes arrivés aux limites de notre algorithme avec deux exemples (le second étant assez exceptionnel en pratique).

4.2.1 Exemple du lac

Lorsque l'on crée une carte contenant un «**lac**» (virtuel), et que l'on place des aérodromes des deux côtés de ce lac, l'algorithme génétique peine à trouver la meilleure solution. La figure 24 donne un exemple d'une telle carte.

Fig. 24 – Schéma de l'exemple du "lac"



Comme le lac est trop grand pour être traversé (l'autonomie étant toujours de 5 cases), le vol doit se dérouler d'un côté ***ou*** de l'autre du lac. En conséquence, dès que l'algorithme génétique cherche à effectuer un crossover entre une solution passant d'un côté du lac et une solution passant de l'autre côté, lors de la recombinaison, l'individu ainsi créé est non viable : le pilote ne peut pas traverser le lac.

D'ailleurs, dès la création de la population initiale, l'algorithme a des problèmes : comme il génère chaque individu en choisissant chaque gène au hasard, il arrive très souvent que l'individu ainsi créé code, à un endroit, une traversée du lac, qui est impossible (l'appareil n'a pas une autonomie suffisante) ; l'individu est donc non viable et l'algorithme doit en créer un autre. Mais lors de la création du nouvel individu, le même problème se pose à nouveau, puisque l'individu est recréé **entièrement**... Il faut donc des heures de calcul à l'algorithme pour trouver des individus viables. Comme ensuite, fréquemment, il arrive que les crossover ne marchent pas, au final on obtient des résultats catastrophiques pour des temps de calcul exorbitants.

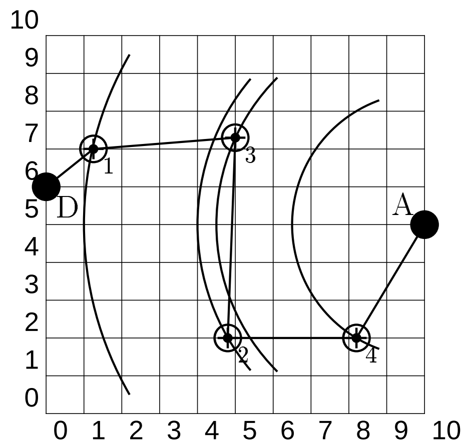
Une parade a été trouvée à ce problème : nous avons scindé en deux la carte, et chaque côté du lac a été considéré comme une nouvelle carte, sur laquelle nous avons fait travailler l'algorithme. Il n'a eu alors aucun mal à trouver quelle était la meilleure solution de chaque côté, puis nous avons comparé les fitness des 2 meilleurs individus, et nous en avons déduit le meilleurs parcours. Il a donc fallu **aider l'algorithme génétique, «à la main»** pour qu'il parvienne à surmonter ses difficultés.

4.2.2 Problème de codage

Enfin, nous avons trouvé un cas très exceptionnel où notre algorithme **ne peut pas** trouver la solution optimale.

La figure 25 présente un exemple pour lequel notre manière de coder les aérodromes du plus éloigné de l'arrivée jusqu'au plus proche ne fonctionnerait pas.

FIG. 25 – PROBLÈME DE CODAGE



Supposons que la distance entre les points 1 et 2 et encore 3 et 4 soit trop grande pour être parcourue sans escale (en prenant par exemple une autonomie de 5.5 cases). Le chemin tracé est alors la seule possibilité qu'a le pilote pour se rendre du point D au point A. L'individu correspondant à ce trajet serait : $S = 1111$, puisqu'il faut se poser sur tous les aérodromes.

Or lorsque l'algorithme va évaluer l'individu S, il comprendra que le pilote doit se poser en 1, puis en 2, puis en 3 et enfin en 4, car c'est dans cet ordre qu'il «lit» le trajet. Et dans cet ordre, le pilote ne peut rallier les aérodromes, car s'il va directement de 1 vers 2, son autonomie sera insuffisante pour atteindre 2...

Ici, le problème est plus sérieux que le problème du lac : cet exemple n'étant pas «linéaire» (puisque le trajet nécessite une sorte de retour en arrière), l'algorithme ne peut trouver la solution. Même si on avait numéroté les aérodromes dans l'autre sens, le problème aurait lieu également.

Si l'on avait voulu que même dans ce cas-là (assez exceptionnel, il est rare qu'en avion on doive «revenir en arrière») l'algorithme trouve la solution, alors tout le codage du problème était à revoir. Des chercheurs ont alors essayé des codages sous formes non plus de bits, mais de couples de bits, ou bien encore de matrices ; mais alors il faut également changer le fonctionnement des crossover (crossover de matrices !)... tout devient beaucoup plus compliqué ! Même la théorie énoncée plus haut, dans le cas de vecteurs de bits, est à revoir !

4.3 Un problème fréquent : Le "Deceptive Problem"

Enfin, un autre phénomène est apparu lors des tests, bien plus fréquent celui-ci. C'est un problème bien connu des chercheurs, qui l'ont baptisé *Deceptive Problem* ou encore *Deception*.

4.3.1 Point de vue théorique

En réalité le *Deceptive Problem* apparait de manière assez claire et même intuitive lorsque l'on a bien compris le principe de fonctionnement d'un algorithme génétique. Nous savons qu'un algorithme génétique se laisse guider au fil des générations par les individus les plus "forts". Ces derniers sont sélectionnés à chaque génération et reproduits dans l'espoir de former des individus encore meilleurs.

Partons donc d'un individu qui paraît très bon au début : il possède une fitness largement supérieure aux autres, et lui, ou ses descendants (encore meilleurs), sont donc reproduits de manière exponentielle au fil des générations. Ainsi l'algorithme «s'oriente» vers une partie de l'espace des solutions, en espérant qu'elle contienne le meilleur individu, la solution optimale.

On commence ainsi à percevoir le problème : que se passe-t-il lorsque la solution optimale est en fait dans un autre endroit de l'espace des solutions ? L'algorithme va se «diriger» vers une région de l'espace contenant une solution **sous-optimale**. Peut-il alors sortir de cette partie contenant la solution sous-optimale pour se diriger vers la partie contenant la solution optimale ? A première vue, ce n'est pas évident car pour cela il faudrait qu'il accepte de créer pendant plusieurs générations consécutives des individus qui se trouvent être moins bons que la solution sous-optimale dans l'espoir de rallier l'autre partie de l'espace des solutions contenant la solution optimale.

Or ceci est «difficile» pour un algorithme génétique. En effet, si lorsqu'il modifie une solution (sous-optimale), l'algorithme obtient des solutions moins bonnes, celles-ci ont tendance à être éliminées par la roulette biaisée...

Nous sommes donc en présence de ce que les américains ont appelé des «Deceptive attractors», terme que nous pourrions traduire de manière abusive par "attracteurs décevants" (expression qui rappelle à juste titre les attracteurs étranges des systèmes chaotiques). Néanmoins nous conserverons le terme de solution sous-optimale, ces deux expressions étant dans ce cas synonymes.

De manière imagée, nous pouvons décrire l'espace des solutions comme un paysage de vallées et de montagnes. Les sommets des montagnes représentent les meilleures solutions (la plus haute montagne est donc la solution optimale), tandis que les vallées représentent l'ensemble des solutions inintéressantes. L'algorithme va donc devoir arpenter ce paysage toujours dans le but de trouver le pic le plus haut. A partir de là, nous pouvons nous représenter le *Deceptive Problem* comme suit : l'algorithme va être plus enclin à emprunter des chemins qui montent faiblement, autrement dit, une montagne large et imposante (facile à repérer). Il n'en reste pas moins que le paysage peut toujours contenir un pic plus étroit, peut-être plus difficile à percevoir à partir de la base car moins imposant, mais pourtant plus haut. Ce pic sera celui qui mènera à la solution optimale et non l'autre.

Lorsque l'algorithme sera arrivé en haut d'une montagne imposante, il aura beaucoup de mal à «sauter» jusqu'au pic le plus haut, car il saute grâce aux mutations, alors qu'il parcourt des chemins continus grâce aux crossover. Etant en haut d'une montagne, tous les chemins continus qui en partent se dirigeront donc vers le bas, et il les négligera. Il ne pourra redescendre de sa montagne et il ne trouvera pas le pic amenant à la solution optimale.

De manière un peu moins intuitive nous pouvons énoncer le même problème à l'aide de l'hypothèse des blocs de construction (voir page 10).Prenons un exemple.

Soient S_1 et S_2 deux schémas dotés d'une *fitness relative* assez grande (deux blocs de construction).

$$\begin{aligned} S_1 &= 111\star\star\star\star\star\star\star \\ S_2 &= \star\star\star\star\star 111\star\star \end{aligned}$$

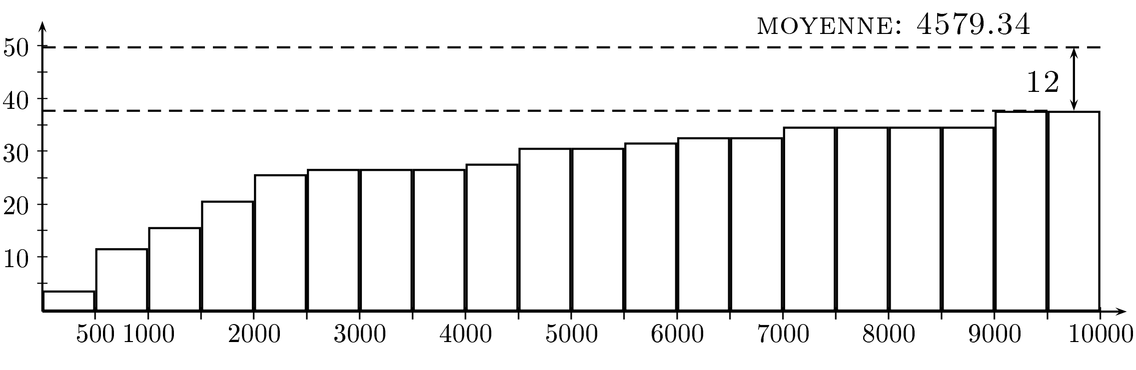
Nous savons que ces deux schémas sont voués à être reproduits de manière exponentielle. Un individu des générations suivantes aura donc de grandes chances de contenir le schéma $S_3 = 111\star\star 111\star\star$ dans son patrimoine. On retrouve alors notre *Deceptive Problem* dans le cas où le schéma $S_4 = 000\star\star 000\star\star$ a une meilleure fitness que le précédent. Il est clair que l'algorithme, suivant l'hypothèse des blocs de construction, aura beaucoup de mal à trouver cette solution sachant qu'il va privilégier au fil des générations des individus correspondant à S_1, S_2 ou à S_3 : le schéma S_4 aura beaucoup de difficultés à se développer dans la population.

Cette explication plus théorique nous montre que le *Deceptive Problem* est un problème de fond des algorithmes génétiques. Et nous avons pu en faire l'expérience dès les premiers résultats.

4.3.2 Mise en évidence du Deceptive Problem

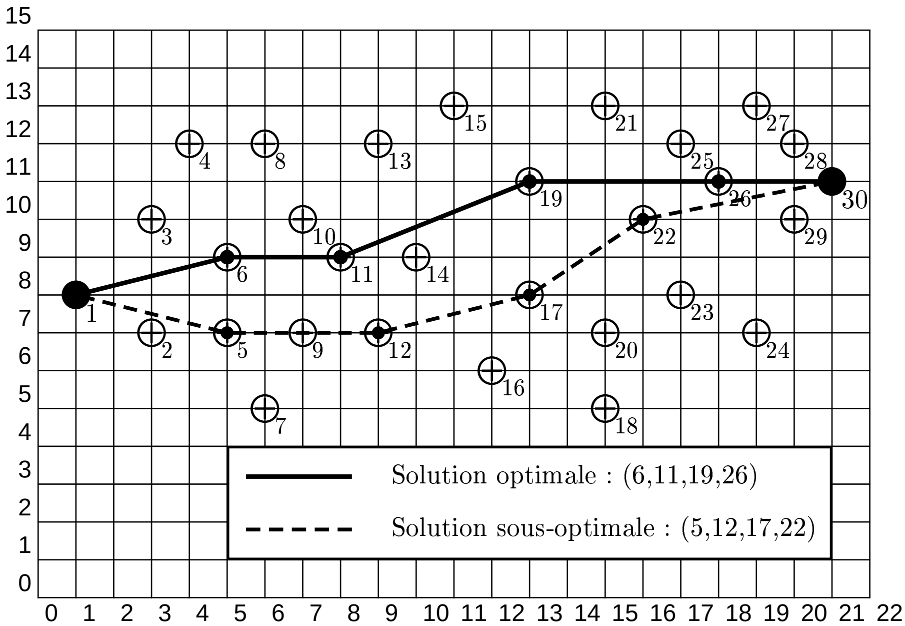
Les effets du *Deceptive Problem* sont en fait visibles sur les graphiques que nous avons présentés dans le chapitre précédent. Considérons à nouveau une série de tests, dont le résultat est présenté figure 26.

Fig. 26 – Paramètres : crossover 0.5, mutation 0.7



Ici, lors de 12 exécutions sur 50, l'algorithme n'a pas su trouver la solution optimale en moins de 10 000 générations. Bien sûr, sur ces 12 fois on est en droit de se demander si la solution qu'il a retenue au bout de la 10 000$^{\text{ème}}$ génération était proche ou non de la solution optimale. Si l'on considère la fitness du meilleur individu de ces 12 tests la réponse est positive.

Fig. 27 – Solutions optimale et sous-optimale



La fitness de la solution optimale (6,11,19,26) est 98.4283404278, tandis que, lors des tests qui ont «échoués» le meilleur individu obtenu au bout de 10 000 générations était (5,12,17,22) et avait une fitness de 97.9858288203. Cet individu est représenté figure 27.

Cet individu constitue en fait notre solution sous-optimale. Il faut avouer que dans ce cas les fitness de ces deux individus ne sont pas très éloignées. Si l'on met à l'écart les cas un peu pathologiques des test réalisés pour des probabilités de crossover ou de mutation valant 1 ou 0, à chaque fois que l'algorithme n'est pas arrivé à trouver la solution optimale au bout de 10 000 générations c'est qu'il a «buté» sur la solution sous-optimale. Pour preuve, les relevés de l'expérience présentés à la figure 28.

Voir figure 28, page suivante

FIG. 28 – APPARITION DU «DECEPTIVE PROBLEM»

PARAMÈTRES : CROSSOVER 0.5, MUTATION 0.7			
test numéro	meilleure fitness	test numéro	meilleure fitness
1	98.4283404278	26	98.4283404278
2	98.4283404278	27	**97.9858288203**
3	98.4283404278	28	98.4283404278
4	98.4283404278	29	**97.9858288203**
5	98.4283404278	30	98.4283404278
6	98.4283404278	31	**97.9858288203**
7	98.4283404278	32	98.4283404278
8	**97.9858288203**	33	**97.9858288203**
9	98.4283404278	34	98.4283404278
10	98.4283404278	35	98.4283404278
11	98.4283404278	36	98.4283404278
12	98.4283404278	37	**97.9858288203**
13	*97.5683397777*	38	98.4283404278
14	98.4283404278	39	**97.9858288203**
15	98.4283404278	40	98.4283404278
16	**97.9858288203**	41	98.4283404278
17	**97.9858288203**	42	98.4283404278
18	98.4283404278	43	98.4283404278
19	98.4283404278	44	98.4283404278
20	98.4283404278	45	98.4283404278
21	98.4283404278	46	**97.9858288203**
22	98.4283404278	47	98.4283404278
23	98.4283404278	48	**97.9858288203**
24	98.4283404278	49	98.4283404278
25	98.4283404278	50	98.4283404278

C'est donc un problème récurrent des algorithmes génétiques, dû, nous l'avons vu, à leur fonctionnement même. Nous l'avons recontré pratiquement lors de chaque série de tests.

En effet lorsque l'algorithme trouve cette solution sous-optimale ayant une très bonne fitness, il la reproduit de nombreuses fois (par le principe de la roulette biaisée) et peine ensuite à s'en éloigner. En fait dans notre exemple, pour trouver mieux que **(5,12,17,22)**, il faut atteindre pratiquement «directement» **(6,11,19,26)**, c'est-à-dire muter les 8 gènes adéquats en même temps ! Ce qui est, statistiquement, très difficile. Lors de nos tests, nous faisions afficher par l'algorithme le meilleur individu de la génération en cours. Au cours de nos centaines de tests, nous l'avons vu une seule fois «revenir» de cette solution sous-optimale pour finalement trouver la solution optimale.

Conclusion

Cette étude nous a amené à bien cerner les avantages et les inconvénients des algorithmes génétiques, grâce aux nombreux tests que nous avons réalisés. Ayant nous-même écrit en CAML un algorithme génétique, nous avons également pu bien comprendre le fonctionnement et le comportement de tels algorithmes.

Avantages :

1. **Rapidité**. Grâce à la puissance de l'ordinateur, l'algorithme génétique crée très rapidement des milliers de générations, ce qui permet réellement de «voir» sous nos yeux l'Evolution agir...
2. **Efficacité**. Même s'il bute parfois sur des solutions sous-optimales, l'algorithme génétique obtient toujours des solutions de très bonnes facture très rapidement, ce qui est son but premier (puisqu'on l'utilise souvent sur des problèmes dont on ne connait pas la solution optimale).
3. **Modularité**. Grâce à de nombreux opérateurs différents, nous pouvons modifier seulement un des sous-programmes (par exemple pour changer le crossover à deux points en crossover à un point ou inversement) et voir immédiatemment le résultat sur le comportement de l'algorithme. De plus de nombreux autres opérateurs (inversion, crossover uniforme,...) peuvent être rajoutés très facilement.

Inconvénients :

1. **Spécificité**. L'algorithme génétique est différent selon le problème que l'on souhaite résoudre, et il faut souvent avoir une première idée de la solution pour «guider» l'algorithme et lui permettre d'être efficace. Il en résulte une mise au point assez longue (choix du codage, des opérateurs efficaces,...)
2. **Complexité**. Au sens informatique du terme, la complexité d'un programme ou d'un algorithme est le nombre d'opérations de base (additions, comparaisons, tests,...) que doit effectuer l'algorithme pour résoudre le problème donné. Or ici nous avons affaire à un **algorithme stochastique** : son comportement est déterminé par l'entrée mais aussi par des valeurs produites par un générateur de nombres aléatoires. En conséquence, c'est un **algorithme heuristique**, c'est-à-dire ne fournissant pas forcément une solution optimale. La complexité n'est donc pas vraiment définie pour ce genre d'algorithmes. Il est alors difficile de le comparer avec d'autres méthodes de résolution, si celles-ci existent...

En conclusion, nous dirons que l'algorithme génétique est un **outil** : il est utile, lorsqu'on a affaire à un problème trop complexe (mettant en jeu trop de variables), pour déterminer les régions de l'espace de solutions contenant de très bonnes solutions à ce problème. Mais il reste finalement confiné à l'état d'outil, ne pouvant fournir de solution optimale, ou s'il la fournit, ne pouvant démontrer qu'elle est optimale.

Ce sont surtout ses capacités à trouver très rapidement de très bonnes solutions qui servent aux ingénieurs : un article de la revue *Pour la science* donne l'exemple d'un ingénieur qui fit appel à un algorithme génétique pour calculer la forme d'une turbine d'avion. Le calcul de cette forme faisant intervenir de nombreux paramètres, il soumit le problème à un algorithme génétique, qui lui indiqua les régions de l'espace de solutions les plus prometteuses ; mais c'est l'ingénieur qui ensuite détermina la forme finale de la turbine. L'algorithme génétique lui permit d'éviter de nombreux calculs intermédiaires fastidieux pour déterminer ces régions de l'espace : il estime qu'il a ainsi gagné une semaine de travail !

Annexe : Liste des programmes

Voici la liste des programmes créés en Caml pour faire fonctionner notre algorithme génétique :

Deux nouveaux types Caml ont été créés avant d'écrire ces programmes :

```
type individu = { mutable trajet   : int vect ;
                  mutable fitness  : float    ;
                  mutable rfitness : float    ;
                  mutable cfitness : float    } ;;


type donnée =
    |Newline
    |Int of int
    |Float of float
    |String of string
    |Int_vect of int vect
    |Trajet of int vect
    |Float_vect of float vect ;;
```

Bibliographie

Voici les ouvrages, articles de journaux, cours et sites Internet qui ont servi de base à notre étude :

- ***Genetic Algorithms + Data Structures = Evolution Programs***, [Ouvrage], Zbigniew MICHALEWICZ, Springer, 1999.
- ***Les algorithmes génétiques***, [Article], John HOLLAND, *Pour la science n° 179*, 1992.
- ***Algorithmes stochastiques***, [Cours], Vincent BERRY, 5 mars 2001.
- ***Les algorithmes génétiques***, [Site Internet], Jean-Marc ALLIOT, *http ://www.recherche.enac.fr/opti/papers/thesis/HABIT/main002.html*.

Index