Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche
researchportal.unamur.be
University of Namur

THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

Algorithmes génétiques et réseaux neuronaux

Soufi, Hammam

Award date:
2001

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 29. Apr. 2024

Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'Informatique
Année Académique 2000-2001

ALGORITHMES GENETIQUES

ET

RESEAUX NEURONAUX

Hammam SOUFI

Promoteur : Jean-Paul Leclercq

Mémoire en vue de l'obtention du grade de Maître en Informatique

UBS 9146817

Abstract

We first investigate the problems concerned with the Genetic Algorithms (GA), especially those for which the GA's can offer solutions, at least, as good as the others problem-solving methods. These problems are, principally, optimization and decision problems (NP-Complete Problems). Then, we review the biological foundations which have inspired the construction of the GA's, like the theory of evolution and the Genetics ; next we consider the Evolutive Programs (EP), that belong to the same family of the GA's, and the Complex Adaptive Systems (CAS) which are the upper class of the GA's and EP's. We study the essential aspects of the GA, both theoretically and with many examples. At the same time, we discuss the implementation of each component part of the GA.

We review the topic of Neural Networks (NN), and then a case study of the combination of a GA and a NN is presented. A detailed case study of the GA is presented. The simulations are carried out : numerical are first collected, then graphical results are displayed and finally they are studied, commented and confronted to the theory. We finish by drawing a conclusion about the GA's.

Résumé

Le travail que j'ai effectué dans ce mémoire ne se résume pas, et en aucun cas, à un enchaînement des paragraphes et des parties des textes dont le seul lien, ou le « p.g.c.d. » est le titre du mémoire. Comme le ver à soie dévore des dizaines et des dizaines de feuilles du mûrier, puis il tisse fil après fil son cocon, nous avons tissé phrase après phrase, et ligne après ligne ce texte qui est entre vos mains.

Ce texte a un corps, un esprit et un âme. Son corps est les Algorithmes Génétiques (AG) et les Réseaux Neuronaux (RN). Son esprit est, d'un côté, les problèmes NP-complets, l'optimisation et la décision, la résolution des problèmes et un paradigme de programmation, et de l'autre côté, la biologie, la génétique, l'évolution et le cerveau humain. Tandis que son âme est : découvrir, apprendre et réflechir.

Au début, nous avons commencé par la finalité et l'opportunité des AG : pour quels types de problèmes fonctionnent-ils bien ou mieux que les autres méthodes de résolution ? Puis, nous avons décrit leurs fondements : d'un côté, la théorie de l'évolution et la génétique qui ont inspiré, surtout, les premiers travaux en algorithmique génétique, et qui continuent à le faire ; et de l'autre côté, les Systèmes Adaptatifs Complexes (SAC) qui sont la sur-classe des AG, et les Programmes Evolutifs (EP) qui sont les cousins des AG, et qui appartiennent, eux aussi, aux SAC. Ensuite nous avons passé en revue, et en détails, les différents composants et techniques des AG, en utilisant la méthode suivante : décrire la théorie, l'illustrer par des exemples et construire l'implémentation en Pascal ou en C/C++.

Nous avons ensuite fait un détour par les Réseaux Neuronaux (RN), où nous avons passé en revue certains des aspects fondamentaux des RN, puis nous nous sommes attardés sur la combinaison des RN avec les AG, et nous avons étudié une application de cette combinaison.

Enfin, nous avons utilisé l'outil de simulation pour étudier les aspects fondamentaux des AG : l'optimisation d'une fonction mathématique

style Black Box, et l'influence des paramètres sur le comportement de l'AG. Nous avons mené des simulations, construit des tableaux et des graphiques, et le plus important, tiré des conclusions et comparé les résultats avec la théorie.

Avant-propos

Quand nous avons commencé la préparation de ce mémoire, nos connaissances sur les algorithmes génétiques étaient fort limitées. Au fil du temps, et de nos lectures, et à travers les dédales des livres, magazines et articles sur l'Internet, nous avons découvert et appris beaucoup de choses concernant les 4 domaines : la génétique, les systèmes adaptatifs complexes, la calculabilité et la complexité, et les algorithmes génétiques bien entendu. Pendant ce périple intellectuel, nous avons appris beaucoup de choses :

Concernant la génétique, il y a eu 2 aspects positifs :

1) l'acte d'apprentissage, en soi, est très enrichissant, d'autant plus que nos connaissances du domaine étaient au départ assez modestes ;

2) le sujet en lui-même est très intéressant, car on dit que le 21^{ième} siècle sera celui de la génétique.

Quant aux systèmes adaptatifs complexes, ils sont importants à deux niveaux : d'une part ils forment la majorité des systèmes réels (comme le système immunitaire, l'Economie, les écosystèmes…), et d'autre part ils constituent un domaine passionant et très intéressant à comprendre.

Puis nous avons effectué une petite visite agréable d'une contrée montagneuse qu'on appelle la complexité et la calculabilité.

Après quoi nous nous sommes amarrés dans le port des algorithmes génétiques. Là, nous y avons appris comment il est possible, avec des méthodes basées sur des principes simples et naturels – apparemment, il y a toujours à apprendre de cette formidable Nature -, résoudre des problèmes complexes avec une précision relativement bonne et surtout avec une rapidité pratiquement imbatable, dans des domaines qui occupent une place importante dans la vie et où on rencontre les problèmes les plus ardus comme les problèmes de décision et d'optimisation.

Enfin, je remercie le Professeur J-P Leclercq pour ses conseils avisés et ses recommandations qui m'ont orientés dans la bonne direction et m'ont ainsi permis d'éviter moults écueils qu'un novice collectionne habituellement au fil de son travail et de ses découvertes dans des domaines relativement méconnus.

Introduction générale

L'évolution est une machine étonnante de résolution de problèmes. Les algorithmes génétiques sont une classe de modèles computationels qui imitent le processus de la sélection naturelle en vue de résoudre des problèmes. Ces routines, à leur tour, donnent aux ordinateurs une plus grande flexibilité pour s'adapter aux circonstances inattendues en analysant des grands ensembles de données, ou en prédisant des résultats incertains.

Ainsi, on peut voir et rencontrer de plus en plus des programmes d'ordinateurs qui " évoluent " comme des organismes vivants, conçoivent des turbines, optimisent des réseaux de communication ou commandent l'acheminement du gaz naturel dans les pipelines.

Les organismes vivants résolvent superbement de nombreux problèmes, et leurs capacités d'adaptation en remontrent aux meilleurs programmes informatiques. De surcroît, les informaticiens passent des mois, voire des années, à construire des algorithmes, alors que les organismes vivnats s'élaborent spontanément grâce aux mécanismes de la sélection naturelle et de l'évolution .
Des chercheurs pragmatiques ont cessé d'envier le remarquable pouvoir de l'Evolution ; ils ont tenté de la reproduire, après avoir compris que la sélection naturelle élimine l'un des obstacles majeurs à la conception des programmes : la spécification préalable de toutes les caractéristiques d'un problème.

En reproduisant informatiquement des mécanismes d'évolution, on "élève" aujourd'hui des programmes qui résolvent des problèmes dont personne ne comprend profondément la structure. D'ores et déjà, ces "algorithmes génétiques" ont à leur crédit des percées dans la conception de systèmes complexes, tels des réacteurs d'avions ou la construction des robots.

D'autre part, il y a des personnes qui s'interrogent et qui demandent pourquoi copier la nature, alors que l'esprit humain débarrassé de tout préjugé et de toute contrainte (si ce n'est celles engendrées par la structure même de son système neuronal) serait à même de concevoir des mécanismes autrement plus compétitifs pour les applications qu'il envisage? La nature, même s'il est clair qu'elle peut aboutir à des structures performantes, n'est certes pas optimale en elle-même et il est sans doute préférable d'éviter des actes de foi béats ou naïfs.

On répond à ces personnes par plusieurs points:
D'abord, leur vision de l'esprit humain est complètement idyllique, et elle porte en elle-même une certaine contradiction. Si on se réfère à la théorie de l'évolution de Darwin, l'être humain est un pur produit de cette évolution de la Nature, cette même Nature qui n'est pas optimale et par conséquent un non-optimal - l'être humain - peut très difficilement- si ce n'est pas impossible - engendrer et créer l'optimal, surtout si on est juge et partie. Deuxièmement, malgré le fait que jusqu'à maintenant les algorithmes génétiques ne bénéficient pas - selon la grande majorité des informaticiens- des bases théoriques aussi solides que les autres méthodologies, ni de la compréhension totale et profonde de leur fonctionnement, ont réussi à résoudre beaucoup de problèmes difficiles et ils ont fourni des résultats au moins aussi bons - si ce n'est pas plus - que les autres algorithmes des méthodes existantes.
Troisièmement, grâce à J. HOLLAND et aux algorithmes génétiques une nouvelle méthodologie de programmation en informatique est née. Ainsi, en plus des méthodologies qui existaient déjà en informatiques comme:la programmation orientée-objet et la programmation fonctionnelle, on a ajouté la programmation évolutive. Aucune de ces précédentes méthodes n'offrait un support total et complet pour la construction d'un programme évolutif. Le but général de cette nouvelle méthodologie est la création des outils appropriés pour l'apprentissage (ici l'optimisation est vue comme un processus d'apprentissage) en utilisant de préférence, une architecture basée sur des processeurs en parallèle.
D'ailleurs, pour ce troisième point et les conséquences qui en découlent -un nouveau paradigme de computation, des nouveaux horizons pour les applications de l'informatique, un autre regard sur les

Introduction générale

L'évolution est une machine étonnante de résolution de problèmes. Les algorithmes génétiques sont une classe de modèles computationels qui imitent le processus de la sélection naturelle en vue de résoudre des problèmes. Ces routines, à leur tour, donnent aux ordinateurs une plus grande flexibilité pour s'adapter aux circonstances inattendues en analysant des grands ensembles de données, ou en prédisant des résultats incertains.

Ainsi, on peut voir et rencontrer de plus en plus des programmes d'ordinateurs qui " évoluent " comme des organismes vivants, conçoivent des turbines, optimisent des réseaux de communication ou commandent l'acheminement du gaz naturel dans les pipelines.

Les organismes vivants résolvent superbement de nombreux problèmes, et leurs capacités d'adaptation en remontrent aux meilleurs programmes informatiques. De surcroît, les informaticiens passent des mois, voire des années, à construire des algorithmes, alors que les organismes vivnats s'élaborent spontanément grâce aux mécanismes de la sélection naturelle et de l'évolution .
Des chercheurs pragmatiques ont cessé d'envier le remarquable pouvoir de l'Evolution ; ils ont tenté de la reproduire, après avoir compris que la sélection naturelle élimine l'un des obstacles majeurs à la conception des programmes : la spécification préalable de toutes les caractéristiques d'un problème.

En reproduisant informatiquement des mécanismes d'évolution, on "élève" aujourd'hui des programmes qui résolvent des problèmes dont personne ne comprend profondément la structure. D'ores et déjà, ces "algorithmes génétiques" ont à leur crédit des percées dans la conception de systèmes complexes, tels des réacteurs d'avions ou la construction des robots.

D'autre part, il y a des personnes qui s'interrogent et qui demandent pourquoi copier la nature, alors que l'esprit humain débarrassé de tout préjugé et de toute contrainte (si ce n'est celles engendrées par la structure même de son système neuronal) serait à même de concevoir des mécanismes autrement plus compétitifs pour les applications qu'il envisage? La nature, même s'il est clair qu'elle peut aboutir à des structures performantes, n'est certes pas optimale en elle-même et il est sans doute préférable d'éviter des actes de foi béats ou naïfs.

On répond à ces personnes par plusieurs points:
D'abord, leur vision de l'esprit humain est complètement idyllique, et elle porte en elle-même une certaine contradiction. Si on se réfère à la théorie de l'évolution de Darwin, l'être humain est un pur produit de cette évolution de la Nature, cette même Nature qui n'est pas optimale et par conséquent un non-optimal - l'être humain - peut très difficilement- si ce n'est pas impossible - engendrer et créer l'optimal, surtout si on est juge et partie. Deuxièmement, malgré le fait que jusqu'à maintenant les algorithmes génétiques ne bénéficient pas - selon la grande majorité des informaticiens- des bases théoriques aussi solides que les autres méthodologies, ni de la compréhension totale et profonde de leur fonctionnement, ont réussi à résoudre beaucoup de problèmes difficiles et ils ont fourni des résultats au moins aussi bons - si ce n'est pas plus - que les autres algorithmes des méthodes existantes.
Troisièmement, grâce à J. HOLLAND et aux algorithmes génétiques une nouvelle méthodologie de programmation en informatique est née. Ainsi, en plus des méthodologies qui existaient déjà en informatiques comme:la programmation orientée-objet et la programmation fonctionnelle, on a ajouté la programmation évolutive. Aucune de ces précédentes méthodes n'offrait un support total et complet pour la construction d'un programme évolutif. Le but général de cette nouvelle méthodologie est la création des outils appropriés pour l'apprentissage (ici l'optimisation est vue comme un processus d'apprentissage) en utilisant de préférence, une architecture basée sur des processeurs en parallèle.
D'ailleurs, pour ce troisième point et les conséquences qui en découlent -un nouveau paradigme de computation, des nouveaux horizons pour les applications de l'informatique, un autre regard sur les

Table des matières

Chapitre1 : Fondements et Taxonomie des Algorithmes Génétiques

1.1. La classe des problèmes intrinsèquement complexes

Quand on est en train de résoudre un problème, on cherche généralement une solution qui est la meilleure parmi tant d'autres. L'espace de toutes les solutions réalisables ou faisables (c-à-d toutes les objets parmi lesquels se trouve la solution recherchée) est appelé **espace de recherche** (ou espace d'états). Chaque point de l'espace de recherche représente une solution réalisable, et chaque solution réalisable peut être " marquée" par sa valeur ou son appropriation - son aptitude - au problème. Quand on cherche notre solution, on recherche en fait un point parmi toutes les solutions réalisables- donc un point de l'espace de recherche.

La recherche d'une solution est équivalente donc à une recherche d'un certain extremum (minimum ou maximum) dans l'espace de recherche. Cet espace peut être totalement connu au moment de la résolution du problème, mais habituellement on connaît quelques points seulement de lui, et on génère les autres points pendant que le processus de la résolution continue à trouver des solutions.

Le problème est que la recherche peut être très compliquée, et de ce fait il y a une grande classe de problèmes intéressants pour lesquels il n'existe pas d'algorithme raisonnablement rapide qui ait été développé.

Beaucoup de ces problèmes sont des problèmes d'optimisation qu'on rencontre fréquemment dans beaucoup d'applications. Etant donné un difficile problème d'optimisation, il est souvent possible de trouver un algorithme efficace qui fournit une solution presque optimale. Pour quelques difficiles problèmes d'optimisation, on peut aussi utiliser les algorithmes probabilistes-ou probabilistiques- ; Ces algorithmes ne garantissent pas la valeur optimale, mais en choisissant des représentants de la population en nombre suffisant (autrement dit, en augmentant la taille de l'échantillon), la probabilité d'erreur peut être rendue aussi petite qu'on veut.

Il existe beaucoup de problèmes pratiques importants d'optimisation pour lesquels de tels algorithmes de haute qualité sont disponibles. Par exemple, on peut appliquer le recuit simulé aux problèmes du routage des fils et du placement des composants dans le design des VLSI(very large scale integration), ou au problème du voyageur de commerce. D'ailleurs, plusieurs autres problèmes d'optimisation combinatoires et de grandes dimensions (plusieurs parmi eux ont été démontrés NP-hard) peuvent être résolus approximativement sur les ordinateurs actuels en utilisant des techniques du type Monte Carlo.

En général, n'importe quelle tâche à accomplir peut être vue comme une résolution d'un problème, qui à son tour, peut être perçu -comme on l'a déjà vu - comme une recherche dans un espace de solutions potentielles. Du fait qu'on recherche "la meilleure" solution, on peut voir cette tâche comme un processus d'optimisation.

Pour des petits espaces, les méthodes classiques exhaustives sont généralement suffisantes ; pour des grands espaces, des techniques spéciales d'intelligence artificielle doivent être employées.
Ces techniques ou méthodes montrent comment on trouve une solution **acceptable ou convenable** (c-à-d non nécessairement la **meilleure solution),** et on peut citer ici quelques unes parmi elles comme la méthode du grimpeur -ou de l'alpiniste- (hill climbing), la méthode de la recherche Tabou, la méthode du recuit simulé et les algorithmes génétiques. En fait, les algorithmes génétiques sont des algorithmes

stochastiques dont les méthodes de recherche modélisent un certain phénomène naturel : l'hérédité génétique et la lutte darwinienne pour la survie.

Le but des AG est de fournir une méthode combinatoire puissante en optimisation (on appèle **problème combinatoire** un problème à nombre fini- mais généralement très grand - de solutions discrètes, obtenues par combinaison d'un nombre fini d'éléments; par exemple, le problème du voyageur de commerce est un problème combinatoire). Pour se rendre compte de la grande difficulté de ce genre de problème, prenons, par exemple, le problème du voyageur de commerce (TSP). On donne un ensemble V de n villes et, pour chaque paire de villes, la distance $d(v_i,v_j)$ de la ville v_i à la ville v_j . On précise une constante b. Le problème est de déterminer s'il existe un parcours fermé (i.e. revenant à son point de départ, ou un tour) des villes dont la longueur est inférieure à b (cf. [WOL 91]).

Mathématiquement, cela revient à trouver une permutation v_{p1} , $v_{p2},\ldots,v_{pn}$ telle que

$$(\Sigma_i\, d(v_{pi}\,,v_{pj})) + d(v_{pn}\,,v_{p1}) = b \text{ , où } 1 = i \leq n \qquad (1)$$

Il y a un algorithme évident pour résoudre ce problème : calculer la somme (1) pour toutes les permutations possibles. Toutefois, ce nombre de permutations est égal à n! et, par conséquent, cet algorithme n'est pas polynomial. Ici on a présenté le (TSP) comme un problème de décision ; Il en existe une version apparentée qui a la forme d'un problème d'optimisation : quelle est la longueur minimale d'un parcours des villes. Pour voir l'équivalence des deux problèmes, il suffit de résoudre le problème d'optimisation et de comparer le résultat b_0 à la constante b.

Donc un tour de 30 villes nécessitera 30! additions. En supposant qu'on peut faire 1 milliard d'additions par seconde, le test de toutes les permutations prendra 8.000.000.000.000.000 années ($\pm\ 8*10^{15}$) années !? Si on ajoute 1 seule ville, cela va augmenter le nombre d'additions en le multipliant par 31 ?! Donc la solution de tester toutes les possibilités est sûrement impossible.

Un algorithme génétique peut être utilisé pour trouver une solution dans un temps beaucoup plus court - plus exactement incomparablement plus court -. Malgré le fait que cet algorithme ne va probablement pas trouver la meilleure solution, néanmoins il peut trouver une solution presque parfaite - autrement dit, très près de La Solution parfaite - en moins d'une minute ?!!

L'algorithmique génétique résout également une autre sorte de problèmes qui résistent depuis longtemps aux méthodes de programmation classique : la détermination d'un équilibre entre l'exploration et l'exploitation. Une fois trouvée une bonne stratégie aux échecs, par exemple, on peut se cantonner à l'application de cette stratégie, mais cette exploitation bloque alors la découverte de nouvelles stratégies plus originales. On n'obtient d'améliorations que si l'on se risque à tester de nouvelles stratégies, mais, comme les tests échouent souvent (avec une moyenne de dix coups possibles à partir d'une position, et à raison de 30 coups par joueur au cour d'une partie, le nombre de stratégies différentes est de l'ordre de 10^{60}, mauvaises pour la plupart), l'exploration dégrade la performance. Tous les systèmes qui s'adaptent et apprennent doivent décider jusqu'à quel point le présent doit être hypothéqué pour le futur. La solution que les algorithmes génétiques apportent à ce problème repose sur le mécanisme de crossing-over (on va traiter ce mécanismes avec plus de détails au chapitre 2).

1.2. Les AG et les problèmes NP-complets

En premier lieu, on va essayer d'interpréter la NP-complétude (cf. Annexe 2.), puis on va parler des problèmes intrinsèquement complexes, et enfin on va montrer les liens évidents entre les AG et ces problèmes. Supposons que l'on soit confronté à un problème pour lequel on recherche une solution algorithmiques et qu'il est établi que ce problème est NP-complet. Sous l'hypothèse que $P \neq NP$, notre

problème n'a donc pas de solution algorithmique polynomiale. Faut-il pour cela définitivement renoncer à le résoudre par un algorithme ? Les arguments développés ci-après montrent que ce n'est pas forcément le cas :

• Nous avons mesuré la complexité en considérant le cas le plus mauvais . L'absence d'un algorithme polynomial indique donc uniquement l'absence d'un algorithme dont le comportement est polynmoial sur **toutes** les instances du problème . Rien n'exclut l'existence d'un algorithme qui a un comportement polynmoal dans 99% des cas .

• Les problèmes NP-complets ont la particulité de n'être solubles qu'en explorant un grand nombre de cas .Une technique permettant d'obtenir des algorithmes souvent (mais pas toujours) efficaces pour ces problèmes est d'utiliser des heuristiques pour limiter le nombre de cas à explorer. Donc plutôt que d'énumérer systématiquement toutes les possibilités , on se sert de critères approximatifs pour découvrir rapidement la solution recherchée. La théorie de la NP-complétude nous apprend que de telles méthodes heuristiques ne peuvent pas **toujours** donner de bons résultats. Toutefois rien n'empêche qu'elles soient très efficaces pour de nombreuses instances d'un problème .

• **Si le problème NP-complet à traiter est un problème d'optimisation , on peut se contenter d'une solution approximative -ce que font les méthodes non traditionnelles comme les AG, les RN, le recuit simulé , ...etc -. C'est-à-dire que plutôt que de chercher *la solution optimale*, on peut chercher une solution proche de l'optimum . Il existe par exemple des problèmes d'optimisation NP-complets pour lesquels on connaît des algorithmes polynomiaux calculant une solution qui ne diffère de la solution optimale que d'un facteur constant.**

• Le problème dont on a démontré la NP-complétude est parfois plus général que le problème qui doit effectivement être traité. Les instances auxquelles on applique réellement l'algorithme peuvent satisfaire des contraintes permettant de trouver une solution polynomiale .

Quant aux problèmes intinsèquement complexes , on peut dire que certains problèmes quoique théoriquement solubles par programmes , ils sont en pratique **infaisables** , car nécessitant des ressources incompatibles avec les réalités physiques . Comme on l'a déjà dit (cf. Annexe 2.), si il existe un algorithme de complexité polynomiale , alors le problème est **pratiquement faisable** , et si il n'existe pas d'algorithme de complexité polynomiale alors le problème est **pratiquement infaisable** - Existence d'algorithmes de complexité exponentielle (ou pire) -.

Beaucoup de problèmes intéressants sont intinsèquement complexes :
- Voyageur de commerce , HC ,VC , ...
- problèmes d'ordonnancement
- problèmes de planification
- problèmes de décision
- ...

Et si on étudie les tableaux suivants on constate que ni l'évolution technologique ni la rerésentation des données n'ont pas une grande influence sur l'efficacité des algorithmes exponentiels , d'où le très grand intérêt des méthodes non classiques comme les algorithmes génétiques , les réseaux neuronaux , ...etc .

Complexité

• Hypothèses
  - ❒ ordinateur 100 Mips
  - ❒ traitement de 1 élément = 100 instructions machine

	10	50	100	500	1000	10000
n	.00001 sec.	.00005 sec.	.0001 sec.	.0005	.001	.01
n^2	.0001 sec.	.0025 sec.	.01 sec.	.25 sec.	1 sec.	1.6 min.
n^3	.001 sec.	.125 sec.	1 sec	2.08 min.	16.6 min.	11.57 jours
n^5	.1 sec.	5.2 min.	2.7 heures	1 année	31.7 années	31×10^3
2^n	.059 sec.	35.7 années	4×10^{14} siècles	...	...	...
3^n		2×10^8 siécles	...	...	...	...

Les algorithmes exponentiels sont intrinsèquement complexes

Complexité	ordinateur d'aujourd'hui	ordinateur 100x	ordinateur 1000x
n	$N_1 = 3.6\times10^9$	$100N_1$	$1000N_1$
n^2	$N_2 = 600000$	$10N_2$	$31.6\ N_2$
n^3	$N_3 = 1530$	$4.64N_3$	$10N_3$
n^5	$N_4 = 81$	$2.5N_4$	$3.98N_4$
2^n	$N_5 = 31$	$N_5 + 6$	$N_5 + 10$
3^n	N_6	$N_6 + 4$	$N_6 + 6$

Peu importe les évolutions technologiques, un problème intrinsèquement complexe ne peut et ne pourra être résolu que pour de petits exemples. (Tableaux repris de [DEV 99])

1.3. L'adaptation et l'adaptabilité

Les programmes d'ordinateur ont tendance à être statiques ; ils commencent au point A et vont au point B, d'une façon déterminée et suivant un parcours bien spécifié. A la base de la plupart de logiciels, les algorithmes déterministes ont démontré leur efficacité, néanmoins les logiciels souffraient d'une manque de quelque chose fondamentale qu'est l'adaptabilité. Autrement dit , la grande majorité des applications sont des entités fixes qui ne peuvent pas s'adapter aux situations qui n'étaient pas prévues par leurs programmeurs.

Par ailleurs, certains problèmes ne peuvent pas être résolus par les programmes déterministes. Si on cherche un paradigme de l'adaptabilité, il ne faut pas regarder plus loin que la biologie. Les êtres vivants qui sont basés sur un ensemble de principes chimiques simples ont montré une adaptation et une flexibilité remarquables pendant des millions d'années dans un environnement continuellement variable. L'implémentation de ces concepts biologiques crée des logiciels qui fournissent des solutions évolutives. Dans certains cas, un algorithme biologique peut trouver des solutions que son concepteur ne les a jamais envisagé ; Cet aspect permet à ces logiciels d'aller plus loin et de dépasser l'imagination de leur créateur humain.

1.4. Les systèmes adaptatifs complexes (SAC)

Un parmi les plus importants rôles qu'un ordinateur peut jouer est de simuler des processus physiques. Quand un ordinateur- ou plus exactement un programme d'ordinateur- imite le comportement d'un système, comme le flux d'air autour de l'aile d'un avion, il nous fournit un moyen unique pour étudier les facteurs qui contrôlent le comportement de ce flux. Le plus important dans la simulation c'est de fournir au programme des données précises qui décrivent le plus fidèlement possible le système étudié.

Pendant les 50 années passées, les ordinateurs ont réalisé quelques grands succès de ce point de vue. Les dessinateurs- the designers- des avions, des ponts et des yachts utilisent, tous, les ordinateurs régulièrement, et d'une façon presque routinière, pour analyser leurs dessins avant de les réaliser matériellement. Pour ces artefacts, on sait imiter parfaitement leurs comportements, en utilisant des équations découvertes depuis un siècle. Quoi qu'il en soit, il existe des systèmes qui sont d'un intérêt crucial pour l'être humain, et qu'on n' a pas réussi à les simuler par l'ordinateur d'une façon exacte. L'économie, les systèmes écologiques, les systèmes immunitaires, l'embryogenèse et le cerveau, tous montrent des complexités qui bloquent significativement toutes les tentatives de les comprendre.

Prenons par exemple les méthodes basées sur des équations- souvent différentielles-, et qui fonctionnent très bien pour étudier les avions; ces méthodes ont une portée beaucoup plus limitée, et nettement moins d'efficacité en ce qui concerne l'Economie ou les systèmes économiques. Un ministre des finances qui demande à l'ordinateur- au système informatique- de mesurer l'impacte d'un changement de la politique financière, ne peut pas s'attendre à la même exactitude qu'un ingénieur peut s'attendre, en demandant à l'ordinateur de mesurer les implications du changement de l'inclinaison de l'aile d'un avion.

Malgré ces disparités et toutes les difficultés, nous entrons dans une nouvelle ère en ce qui concerne notre capacité de comprendre et d'utiliser ces systèmes très complexes. Les bases de cet optimisme viennent de deux nouveaux progrès.
Le premier, c'est que les scientifiques ont commencé à dégager un noyau commun de ces systèmes : chacun de ces systèmes contient une "structure évolutive" similaire à celles que les autres possèdent, ce qui signifie que ces systèmes se changent et réorganisent leurs composants, pour s'adapter aux problèmes posés par leurs entourages. Ce fait est la raison principale qui rend ces systèmes difficiles à comprendre, et à être contrôler; et ce qu'il fait d'eux un "objectif mobile"- donc difficile à atteindre-. Nous avons, malgré tout cela, appris que les liens et les ressemblances entre ces systèmes sont beaucoup plus forts que des observations superficielles laissent suggérer. Ces mécanismes- d'adaptation- et les similarités profondes de ces systèmes sont assez importants pour qu'on les regroupe sous un nom commun **les systèmes adaptatifs complexes (SAC)** .

Le deuxième progrès important est l'avènement d'une nouvelle ère en calcul (computation). Ce progrès va permettre aux experts- qui ne sont pas nécessairement des spécialistes en informatique- de faire des tests (du type "flight -test") des modèles particulier des systèmes adaptatifs complexes (SAC). Par exemple, un homme politique ou tout autre décideur peut examiner la faisabilité d'un modèle sans nécessairement connaître le code qui est derrière lui. Le même décideur peut de nouveau formuler et tester des autres politiques sur ce modèle, toujours sans s'en soucier du code, et ainsi il peut développer une intuition "informée" des futurs effets de ces politiques. Les modèles basées sur les SAC offrent les possibilités de la formulation des nouvelles politiques qui concernent des problèmes importants et aussi divers que la Balance Commerciale ou le SIDA .

1.5. Les systèmes adaptatifs biologiques

Pour arriver à une compréhension profonde des SAC (comprendre qu'est-ce qu'il les rend complexes et fait qu'ils sont adaptatifs), il est utile d'examiner un système particulier. Considérons le système immunitaire. Il est constitué d'un grand nombre d'unités très mobiles appelées anticorps, qui repoussent- ou détruisent- continuellement une caste d'envahisseurs- les bactéries et les biochimiques- qui se changent toujours, et qu'on les appelle antigènes. De fait que les envahisseurs agissent sous presque une infinité de formes variées, le système immunitaire ne peut pas simplement développer une liste de tous les envahisseurs possibles ; et même s'il prenait le temps de la faire, il n'y aurait pas assez de place pour stocker toutes ces informations. Au lieu de cela, le système immunitaire doit se changer et adapter ses anticorps pour qu'ils soient aptes ("fit to") chaque fois que des nouveaux envahisseurs apparaissent. C'est cette capacité de s'adapter qui fait que ces systèmes sont difficiles à simuler.

Le système immunitaire doit affronter une complication supplémentaire qui est de distinguer lui-même des autres. En effet, le système doit distinguer les parties légitimes de son propriétaire, de l'ensemble des envahisseurs qui se changent continuellement. C'est une tâche herculéenne car les cellules du propriétaire et leurs constituants biochimiques se comptent par des dizaines de milliers d'espèces. Des erreurs d'identification arrivent chez quelques individus, donnant ainsi les maladies connues d'auto-immunité, mais qui sont, fort heureusement, rares. Le système immunitaire est assez performant dans l'auto-identification au point que, pour le moment, il est le meilleur moyen scientifique pour définir l'individualité. Un système immunitaire, par exemple, ne confondra même pas ses propres cellules de celles d'une greffe épidermique venant d'un frère ou d'une soeur.

Comment le système immunitaire gère le processus continu de l'adaptation qui lui permet d'atteindre un niveau d'identification remarquable ? On ne sait réellement pas, ceci malgré l'existence de quelques conjectures avec des degrés différents de certitude. Les modèles de ce SAC sont difficiles à construire. C'est particulièrement difficile d'avoir de l'expertise dans un domaine dont les modèles permettent "des expériences mentales" ; ces modèles qui mettent l'expert à même de développer une intuition sur les différents mécanismes et organisations.

On rencontre des problèmes similaires quand on traite les autres SAC. Chacun d'eux implique un très grand nombre de parties, et subit un ensemble kaléidoscopique d'interactions simultanées. Ces systèmes ont clairement trois caractéristiques communes : **l'évolution, le comportement d'agrégation et l'anticipation**. A travers le temps, les parties évoluent d'une façon darwinienne, en essayant d'améliorer la capacité de leur espèce à survivre dans leur interactions avec les parties qui les entourent. Cette capacité des parties à s'adapter ou à apprendre est la caractéristique centrale des SAC. Certains systèmes adaptatifs sont très simples : un thermostat, qui en allumant ou en arrêtant un calorifère essaye de maintenir la température ambiante constante, donc le thermostat s'adapte- par allumage ou arrêt- continuellement à la température de l'entourage. Quoi qu'il en soit, les processus adaptatifs qui nous intéresse ici sont complexes, car ils impliquent plusieurs parties, et des critères que chacun d'entre eux a un large domaine de, variation (analogue à la température constante recherchée par le thermostat), pour que ces processus fournissent "un bon résultat".

Les SAC montrent aussi un comportement d'agrégation- ou un comportement agrégationnel- qui n'est pas dérivé simplement des actions des parties. Pour le système immunitaire ce comportement agrégationnel est sa capacité à distinguer lui-même des autres, tandis que pour l'Economie cela concerne des domaines qui varient du PNB jusqu'au réseau entier de l'offre et de la demande ; en ce qui concerne l'écologie, ce comportement concerne habituellement le réseau entier de la nourriture, ou les modèles de flux de l'énergie et des matériaux; Pour l'embryologie, c'est le cycle complet du développement d'un individu ; pour le cerveau, c'est le comportement manifeste et apparent qui

l'évoque et qui est contrôler par lui. Généralement, c'est ce comportement agrégationnel qu'on veut comprendre et modifier, et pour cela on doit comprendre comment le comportement agrégationnel **émerge** des interactions des parties.

Et comme si cela n'est pas assez complexe, il y a en plus une caractéristique qui rend ces systèmes encore plus complexes et qu'elle est leur anticipation. En cherchant à s'adapter au changement du contexte, les parties peuvent être considérées comme des règles de développement qui anticipent les conséquences de certaines réponses. Au niveau le plus simple- , l'anticipation n'est pas trop différent du conditionnement pavlovien : "si la cloche sonne, alors la nourriture va apparaître". Quoi qu'il en soit, même pour du simple conditionnement, les effets deviennent bien complexes quand un grand nombre de parties sont conditionnés en différentes façons. Ceci est particulièrement le cas quand les différents conditionnements dépendent des interactions entre les parties. Plus encore, l'anticipation résultante peut causer des changements majeurs dans le comportement agrégationnel, même si par après ils ne se réalisent pas.

L'anticipation d'une pénurie de pétrole, même si cette pénurie ne s'est pas passée, peut causer une grande hausse des prix, et une augmentation des tentatives cherchant des sources alternatives de l'énergie. Cette capacité émergente à anticiper est une des caractéristiques des SAC qu'on comprend le moins malgré qu'elle est parmi les plus importantes.

Il y a finalement un point plus technique qui nécessite que l'on considère . Vu que les parties individuelles d'un SAC révisent continuellement leurs règles "conditionnées" de l'interaction, chacune de ces parties est impliquée par ou noyée dans des environnements perpétuellement renouvelés (le changement du comportement des autres parties). Comme résultat, le comportement agrégationnel du système est habituellement loin d'être optimal, si jamais une vraie optimalité peut être définie pour le système tout entier.

Pour cette raison, les théories standards en physique, en économie et ailleurs, sont peu utiles car ils focalisent sur les points finaux optimaux, alors que les SAC "n'arriveront jamais là". Ils continuent à évoluer, et ils montrent régulièrement des nouvelles formes du système émergent. L'histoire et le contexte jouent un rôle critique, compliquant ainsi, encore plus, la tâche de la théorie et de l'expérience. Bien que quelques parties du système se fixent temporairement dans un optimum local, ils sont habituellement "morts" ou inintéressantes si ils restent dans cet état d'équilibre pour longtemps. C'est le processus du devenir plus que celui des points finaux jamais atteints, qu'on doit étudier si on veut gagner de perspicacité.

- **Les ordinateurs massivement parallèles (OMP)**

L'introduction de l'ordinateur digital a profondément changé notre vision de ce que peut être calculé; et les OMP - des ordinateurs qui contiennent des centaines de milliers de micro-ordinateurs interconnectés- vont , eux aussi, la changer très profondément. Ce n'est pas seulement une question de vitesse, malgré que celle-ci est importante, car les OMP peuvent s'occuper simultanément d'un grand nombre d'actions, et ils fournissent des nouvelles façons d'envisager et d'interagir avec les données.
Ils fournissent aussi des nouvelles voies pour étudier les SAC nettement au-delà de la limite d'une station de travail, et proportionnellement autant que la station de travail est au-delà de la limite d'une calculatrice -ou une règle à calcul.

En effet, les OMP vont produire une révolution dans les recherches sur les SAC comparable à la révolution produite en biologie par l'introduction du microscope. Les effets à long terme du parallélisme massif ne sont pas faciles à prévoir surtout au stade actuel, mais un peu de réflexion offre quelques indications. Au début de l'ère de l'ordinateur,- dans les années '40 et début des années '50-, la plupart des informaticiens prévoyaient un accroissement dans la vitesse et la capacité du stockage, avec en même temps, une augmentation sans limite de la capacité d'aborder les problèmes scientifiques et de business. Mais l'ampleur de ces développements, quand ils sont arrivés, ensemble avec la baisse des

prix qui les a accompagnés, nous a tous étonné et fort surpris. Ces développements ont rendu possible l'utilisation à très grande échelle du traitement de texte, du courrier électronique, des stations de travail personnelles et un ensemble des activités qui y sont attachées, comme les jeux de vidéo personnels et les simulations- flight simulator, ...-.

Ceci a produit des nouveaux secteurs majeurs de l'économie et a altéré en même temps le travail et le Jeu- les loisirs- d'un grand nombre de personnes. Ce processus des développements rapides en vitesse et en capacité- de stockage- accompagnés par des baisses des prix, est pour le moment en cours concernant les OMP. Le nouveau "microscope" sera prochainement aussi pénétrant et répandu que la station de travail aujourd'hui.

1.6. Les modèles des SAC et leurs simulations

Un SAC ne possède pas une seule équation gouvernante, ou une règle qui contrôle le système; Mais au lieu de cela, il possède plusieurs parties distribuées qui réagissent réciproquement, et qui a peu ou rien de commun avec les systèmes à contrôle central. Chacune des parties est gouvernée par ses propres règles ; chacune de ces règles peut participer à l'influence sur le résultat, et chacune peut influencer les actions des autres parties. La structures résultante qui est basée sur des règles devient une brique de base - une graine- des procédures évolutives qui permettent au système de s'adapter à son environnement. Nous pouvons acquérir une meilleure compréhension de ces procédures évolutives si nous regardons d'abord de plus près ce concept de structure distribuée et qui est basée sur des règles.

La plupart des règles peuvent être réduites à des simples règles **condition / action** : **Si** la [condition est vraie ou vérifiée] **alors** exécute l'[action] .Les plus simples règles de cette forme ressemblent beaucoup à des spécifications des réflexes psychologiques : **si** [la surface sent chaude], **alors** exécute [un retirement brusque de la main] ; **si** [il y a un objet qui se déplace rapidement dans le champ visuel périphérique] **alors** exécute [un mouvement des yeux jusqu'à que l'objet soit dans le centre du champ visuel]. Des règles plus compliquées agissent par des messages envoyés par des autres règles, et qui à leur tour envoient leurs propres messages :
Si [il y a message X], **alors** exécute [une transmission du message Y]. Des activités beaucoup plus compliquées peuvent être exprimées par des combinaisons des pareilles règles ; en fait tout calcul qui peut être exprimé- décrit- dans un langage de programmation peut être exprimé par une combinaison appropriée de règles condition/action.

Cette organisation distribuée basée sur plusieurs règles exige des grandes qualités et place des fortes conditions à la simulation sur ordinateur des SAC. L'approche la plus directe est de fournir une simulation dans laquelle plusieurs règles sont actives simultanément- c'est "naturel" pour le calcul massivement parallèle. Quand plusieurs règles actives simultanément, un système distribué, basé sur des règles peut manier des perpétuelles nouveautés.

En rencontrant une situation nouvelle, comme "une voiture rouge se trouvant au bord de la route avec un pneu crevé", le système active plusieurs règles concernées, comme celles de "rouge", "voiture", "pneu crevé", et ainsi de suite. Il construit une "image" de la situation à partir des parties plutôt que la traiter comme une chose entière monolithique qu'il ne l'a jamais auparavant rencontré. L'avantage est similaire à celui obtenu quand quelqu'un décrit un visage en termes de ses parties composantes, plutôt que le traiter comme une entité entière indécomposable. On choisit, disons, 8 composants du visage : cheveux, front, sourcils, yeux, pommettes, nez, bouche et menton. Si on permet 10 variantes- ou variations- de chacune de ces parties composantes - plusieurs couleurs des cheveux, plusieurs formes des fronts, et ainsi de suite-, alors 10^8 = 100.000.000 visages peuvent être décrits en combinant ces composants en différentes façons. D'ailleurs, quand un bloc de construction est utile dans une combinaison, il est au moins plausible qu'il sera utile dans des autres combinaisons similaires. Les blocs

de construction donnent au système la capacité de transférer l'expérience passée aux nouvelles situations.

Le parallélisme massif est clairement un avantage dans la simulation des SAC conçus en termes des règles agissantes simultanément. Un processeur individuel peut être allouer à chaque règle, tandis que les connections entre les processeurs fournissent les interactions entre les règles. Le modèle résultant est à la fois naturel et traité rapidement. Pour subvenir aux besoins de l'adaptation, le système doit avoir des moyens de changer ses propres règles. Telles procédures donnent au système sa caractéristique " la structure évolutive". Il existe deux sortes de procédures computationnelles qui sont importantes : les procédures **d'attribution du crédit** et les procédures de **la règle de découverte**.

L'attribution du crédit - ou des points- est nécessaire car on veut que le système et ses règles évoluent **vers** quelque chose. L'attribution du crédit exige d'abord de définir- de donner un sens à- ce qui est une "bonne" performance, puis elle exige un moyen de choisir et "récompenser" les parties du système qui sont apparemment la cause de la bonne performance. Un système qui récompense la bonne performance peut de ne jamais être optimal, mais il peut s'améliorer de plus en plus. L'attribution du crédit est un problème classique dans le domaine de la recherche en intelligence artificielle. Dans un système basé sur des règles, l'objectif est d'assigner un crédit- un point- aux règles individuelles en fonction de leur contribution à la performance du système entier- agrégé-. On peut voir ce crédit comme un **renforcement** attribué à la règle : plus la règle contribue à la bonne performance, plus forte qu'elle devient, et vice versa. Par "plus forte" on entend que en se basant sur ses succès antérieurs, la règle aura plus de poids dans les futures décisions. Avec les situations successives rencontrées, les règles les plus appropriées entrent en compétition pour le contrôle du comportement du système, et les règles les plus fortes ont le plus de chance de succès. C'est-à-dire, si une règle a produit un bon résultat dans le passé, alors il est très vraisemblable qu'elle sera utilisée dans situations similaires dans le futur.

L'attribution du crédit permet au système de choisir la meilleure des règles qu'il possède, mais il ne peut pas fournir au système des nouvelles règles. Si il doit évoluer en vue de traiter des nouvelles situations, le système doit créer des nouvelles règles. Pour cela le système demande- exige- une certaine procédure pour découvrir des règles. La découverte des règles est un processus subtile, car il est important que le processus de découverte génère des règles **plausibles;** Ces règles qui ne sont pas- vu l'expérience du passé- sûrement fausses. Le philosophe C.S.Pierce est bien instructif à ce sujet. Afin de pouvoir appliquer le raisonnement de Pierce à ce modèle, il est commode de penser que les règles sont construites à partir des petites pièces, ou blocs de construction. Selon l'avis de Pierce, la découverte et la recombinaison des blocs de construction est une étape importante en vue d'assurer la plausibilité des règles nouvelles inventées.

Pour aborder la règle de la découverte en termes de blocs de construction, il est utile de penser des règles au "breeding"- l'élevage ou la reproduction - des règles fortes. Cela signifie que les règles fortes- ou plutôt qui sont devenues fortes grâce à leurs succès- sont sélectionnées comme "parents", et les nouvelles règles descendantes "enfants" sont produites par croisement "**crossing**" des parents. On suppose ici que les règles fortes possèdent à l'intérieur d'elles des blocs de construction de hautes valeurs, et qu'elles doivent les incorporer aux nouvelles règles. Ce processus imite le processus où un éleveur fait croiser des chevaux, ou un fermier produit des nouvelles variétés de maïs hybride. Ici l'objectif est de produire des règles descendantes qui sont équivalentes à des nouvelles hypothèses plausibles.

Les procédures de la règle de découverte- d'exploration- de ce type sont appelées **Algorithmes génétiques**. Un algorithme génétique "apprend" automatiquement en influençant et dirigeant les futures générations de règles vers des combinaisons de blocs de construction d'un niveau au-dessus de la moyenne (comme en génétique, les ensembles de gènes qui s'adaptent simultanément, apparaissent plus fréquemment dans les générations successives). On peut prouver que les algorithmes génétiques trouvent

et recombinent des blocs de construction utiles. Ils ont leurs contreparties dans chacun des SAC connus. Bien sûr, plusieurs des nouvelles règles générées par ce processus sont absurdes- inutiles-, mais les règles absurdes- ou insensées- ne favorisent pas du "bon" comportement, et elles sont systématiquement éliminées.

Cette procédure de la découverte, elle aussi, se prête très bien au calcul massivement parallèle. Le croisement des parents forts est une opération simple qui ne demande qu'un faible traitement par l'ordinateur. Le parallélisme est exploité facilement dans ce cas car l'ensemble de règles dans sa totalité peut être traité comme une population, et l'accouplement- le raccordement- se déroule simultanément dans cette population.

Pour résumer, on peut dire que les composants de base des SAC sont vus comme des ensembles de règles. Les systèmes s'appuient sur trois mécanismes clés:
Le parallélisme, la compétition, la recombinaison. Le parallélisme permet au système d'utiliser les règles individuelles- ou individuellement- comme blocs de construction, et ainsi il active des ensembles de ces règles pour décrire et agir sur les situations variables et changeantes. La compétition permet au système de ranger et de rassembler ces règles selon que la situation l'exige, fournissant ainsi la flexibilité et le transfert d'expérience. Cela est vital dans des environnements réels où l'agent reçoit un torrent d'informations, la plupart d'elles sont non pertinentes, et n'ont aucun rapport avec les décisions courantes. Les procédures d'adaptation- assignation des points et règles de la découverte- extraient et isolent les événements utiles et répétitifs de ce torrent, en les incorporant comme des nouveaux blocs de construction. Enfin, la recombinaison joue un rôle clé dans le processus de la découverte, générant ainsi des nouvelles règles plausibles à partir des règles déjà testées. Elle implémente l'heuristique qui dit que les blocs de construction qui étaient utiles dans le passé, ils seront aussi utiles dans des contextes similaires.

En général, ces mécanismes permettent à un SAC de s'adapter, pendant qu'il utilise les capacités existantes pour répondre, à tout instant à son environnement. En faisant ainsi, le système **établit l'équilibre et préserve la balance entre l'exploration** (l'acquisition de nouvelles informations et de nouvelles capacités) **et l'exploitation** (l'utilisation efficace des informations et des capacités déjà disponibles). Le système qui en résulte est bien fondé en termes de calcul- en termes computationnelles-, et il s'améliore effectivement en atteignant ses buts dans un environnement qui est en perpétuel renouvellement.

Les simulations des SAC exécutées sur ordinateur produisent un flux de données. Les résultats de ces simulations évoquent et rappellent les premiers jours du traitement-batch (traitement par fournées) par les ordinateurs: quand l'output apparaissait comme des interminables pages imprimées et des tableaux numériques, il était difficile de relever les informations importantes ou les interactions surprenantes, mais toutefois beaucoup moins que de réagir à eux. Le rôle de l'utilisateur était réduit plus à observer qu'à expérimenter ou contrôler, ce qu'il ne voulait évidemment pas.

Si on veut rendre les simulations parallèles- ou en parallèle- des SAC accessibles, deux critères doivent être satisfaits : le premier, c'est que la simulation parallèle doit imiter directement les interactions parallèles courantes des SAC. Le second est qu'il faut qu'il existe pour l'utilisateur une interface visuelle, orientée-jeux (elle ressemble aux interfaces des jeux-vidéo), et qui permet aux experts - inhabitués à l'utilisation des ordinateurs pour l'exploration des systèmes-, de contrôler ces systèmes d'une façon simple et naturelle. Par exemple, un homme politique ou un décideur doit être capable d'essayer et d'expérimenter un modèle économique de la même façon qu'un pilote s'entraîne sur un simulateur de vol. Les actions et les décisions doivent être prises comme d'habitude sans aucune connaissance des calculs sous-jacents. Il faut aussi qu'il soit facile de voir si le modèle se comporte de manière réaliste dans des situations bien connues. Cela ajoute une valeur supplémentaire, en permettant aux experts de donner un feedback de la conformité au réel aux concepteurs- designers- des simulations.

Les SAC peuvent être simulés sur des ordinateurs massivement parallèles en définissant un réseau des composants basés sur des règles et qui s'interagissent .En cherchant des phénomènes subtils et profonds dans de telles expériences, on peut implémenter le cycle classique de : hypothèse-test-révision, en vue d'étudier les SAC. La partie expérimentale de ce cycle est particulièrement importante, car ces systèmes fonctionnent typiquement loin de l'état d'équilibre, en opérant continuellement des révisions et des améliorations. Ils **ne se prêtent pas aux approches mathématiques basées sur l'équilibre et sur la linéarité, les attracteurs, les points-fixes ou des méthodes similaires.** Un nouveau cadre mathématique est nécessaire ; un qui tient en compte de la continuelle adaptation, en utilisant la recombinaison des blocs de construction.

Sans un tel cadre, les expériences réalisées sur l'ordinateur seront un peu meilleures que des incursions sans coordination dans des domaines complexes et sans fin. Avec un tel cadre, on peut beaucoup mieux approfondir notre compréhension de ces questions difficiles et importantes.

1.7. Les programmes évolutifs

On utilise le terme commun **Programmes Evolutifs (PE)** pour qualifier tous les systèmes basés sur l'évolution. La structure d'un programme évolutif est la suivante :

```
Procédure Programme Evolutif
begin
 t ← 0
 Initialise P(t)
 Evaluate P(t)
  while ( not termination-condition ) do
    begin
     t ← t+1
     Select P(t) from P(t-1)
     Recombine P(t)
     Evaluate P(t)
    end
end .
```

Figure 1 : La structure d'un programme évolutif

Le programme évolutif est un algorithme probabiliste qui maintient une population d'individus, P(t) = $\{x^t_1,...,x^t_n\}$ à l'itération t. Chaque individu représente une solution potentielle du problème qu'on est en train de résoudre, et dans n'importe quel PE il est implémenté comme une certaine structure - parfois complexe- de données S. Chaque solution x_i^t est évaluée pour mesurer d'une certaine façon son degré " d'appropriation" (fitness). Puis, une nouvelle population (itération t+1) est formée par la sélection des individus les appropriés (étape de sélection). Quelques membres de la nouvelle population subissent des transformations (étape de recombinaison) par les moyens des opérateurs "génétiques" en vue de former des nouvelles solutions. Ils existent des transformations unaires m_i (type mutation), qui créent des nouveaux individus par un petit changement dans un seul individu (m_i : S→ S), et des transformations d'ordre supérieur C_j (type croisement), qui créent des nouveaux individus par la combinaison des parties à partir de plusieurs (2 ou plus) individus (C_j : S x...x S → S). Après un certain nombre de générations, le programme converge - en espérant que le meilleur individu représentera la solution optimale.

Considérons un exemple général. Supposons qu'on doit chercher un graphe qui doit satisfaire certaines exigences (disons, qu'on cherche la topologie optimale d'un réseau de communications qui respecte certains critères : le coût des messages envoyés, la fiabilité, etc). Chaque individu dans le programme de l'évolution représente une solution potentielle du problème, c-à-d, chaque individu représente un graphe.

La population initiale des graphes P(0) (soit elle est générée aléatoirement, soit elle est créée comme le résultat d'un certain processus heuristique)est un point de départ (t = 0) pour le programme évolutif (le programme de l'évolution). La fonction d'évaluation est donnée normalement - elle incorpore les exigences du problème . Cette fonction fournit l'appropriation de chaque graphe, en distinguant ainsi entre les bons et les moins bons individus . Plusieurs opérateurs de mutation peuvent être désignés pour transformer un seul graphe. Quelques opérateurs de croisement peuvent être considérés et qui combinent les structures de deux graphes (ou plus) en une. Le plus souvent tels opérateurs incorporent la connaissance spécifique au problème. Par exemple, si le graphe qu'on cherche est connexe et acyclique (c-à-d, un arbre), un opérateur possible de mutation peut effacer une arête du graphe, et ajouter une nouvelle arête pour connecter deux sous-graphes disjoints. L'autre possibilité sera de désigner une mutation indépendante du problème et qui incorpore cette exigence dans la fonction d'évaluation, en pénalisant les graphes qui ne sont pas des arbres.

Clairement, plusieurs programmes évolutifs peuvent être formulés pour un problème donné. Tels programmes peuvent différer entre eux en plusieurs façons :
Ils peuvent utiliser des différents structures de données pour implémenter un individu singulier ; des opérateurs "génétiques" pour transformer les individus ; des méthodes pour créer la population initiale ; des méthodes pour traiter les contraintes du problème, et les paramètres (la taille de la population) ; les probabilités d'application des différents opérateurs, etc). Toutefois, ils partagent en commun un principe : c'est une population d'individus qui subissent certaines transformations, et pendant ce processus d'évolution- par les transformations- les individus se battent pour leur survie .

L'idée de la programmation évolutive n'est pas nouvelle, et elle était abordée 30 ans passés déjà . Malgré cela, le concept actuel des programmes évolutifs est différent de celui proposé antérieurement. L'actuel est basé entièrement sur l'idée des algorithmes génétiques, mais la différence entre les deux est que maintenant- en PE- on peut utiliser **n'importe quelle structure de données** (c-à-d la représentation du chromosome) appropriée pour un problème ensemble avec **n'importe quel groupe d'opérateurs "génétiques"** , tandis que les algorithmes génétiques utilisent des chaînes binaires de longueur fixe (comme un chromosome, une structure de données S) pour ses individus, et les deux opérateurs : la mutation binaire et le croisement binaire. En d'autres mots, la structures d'un algorithme génétique est la même que pour le programme évolutif (Figure1), et les différences sont cachées au niveau le plus bas.

Chaque chromosome ne doit pas nécessairement représenté par une chaîne de bits, et le processus de recombinaison contient des autres opérateurs "génétiques" appropriés pour la structure donnée et le problème posé. On voit qu'une représentation "naturelle" d'une potentielle solution pour un problème donné, plus une famille d'opérateurs "génétiques" applicables peuvent être bien utile dans l'approximation des solutions de plusieurs problèmes, et cette approche de modèle "naturel" (la programmation évolutive) est une direction prometteuse pour la solution des problèmes en général.

Plusieurs chercheurs ont déjà exploré l'utilisation des autres représentations comme la liste ordonnée- triée- (pour l'empaquetage des boîtes ou des compartiments), les listes imbriquées (pour l'ordonnancement des problèmes dans les usines), les listes aux éléments variables (pour les plans des semi-conducteurs). Durant les années passées, plusieurs applications- des variations spécifiques sur les algorithmes génétiques ont été réalisées. Ces variations contiennent des chaînes de longueurs variables (parmi elles les chaînes dont les éléments sont les règles *si - alors - sinon*), des structures plus riches que les chaînes binaires (par exemple, des matrices et des expériences avec des opérateurs génétiques modifiés pour satisfaire les besoins des applications particulières). On a aussi utilisé un algorithme génétique qui utilise la propagation-arrière (une technique d'apprentissage avec un réseau neuronal) comme opérateur, ensemble avec la mutation et le croisement qu'on utilise pour tailler - limiter - le domaine du réseau neuronal.

D'autres chercheurs ont décrit un algorithme génétique qui exécute une phase dans le processus du dessin- designing- d'un réseau de communication avec packet-switching; la représentation utilisée

n'était pas binaire, et cinq opérateurs (basés sur la connaissance, statistiques, numériques) étaient utilisés. Ces opérateurs étaient bien différents de ceux de la mutation et du croisement. Plusieurs implémentations non-standard seraient créées pour résoudre des problèmes particuliers- simplement, parce que les AG classiques étaient difficiles pour être appliqués directement au problème, et quelques modifications dans les structures du chromosome étaient nécessaires.

En outre, on peut se demander, par exemple, si une stratégie d'évolution est un algorithme génétique? Est-ce que l'inverse est vrai ? Pour éviter toute discussion liée à la classification ds systèmes évolutifs, on les appelle tous simplement "programmes évolutifs" (**PEs**). Pourquoi on est parti des algorithmes génétiques vers des programmes évolutifs plus flexibles ? Même si c'est bien théorisé, les AG ont échoué à fournir des applications réussies en plusieurs domaines. Il paraît que le facteur majeur derrière cet échec est le même que celui qui est responsable de leurs succès : l'indépendance du domaine. Une des conséquences de la simplicité des AG (dans le sens de leur indépendance vis à vis de leur domaine) est leur incapacité à traiter les contraintes non-triviales. Dans la plupart des travaux menés sur les AG, les chromosomes sont des chaînes de bits- listes de 0s et 1s . Une question importante à considérer en dessinant le chromosome qui représentera les solutions d'un problème est l'implémentation des contraintes sur les solutions .

En programmation évolutive, le problème de la satisfaction de la contrainte a une autre saveur. Il n'est pas le résultat de la sélection d'une fonction d'évaluation avec certaines pénalités, mais plutôt la sélection de la "meilleure" représentation chromosomique des solutions, ensemble avec des opérateurs génétiques sensées pour satisfaire toutes les contraintes imposées sur le problème. N'importe quel opérateur génétique doit passer une certaine structure caractéristique des parents à leurs enfants-descendants-, de sorte que la structure de la représentation joue un rôle important dans la définition des opérateurs génétiques.

D'ailleurs, différentes structures de représentation ont différentes caractéristiques de convenance pour la représentation des contraintes, ce qui complique plus encore le problème. Ces deux composantes (représentation et opérateurs) s'influencent mutuellement; On voit que n'importe quel problème demande une analyse prudente qui donnerait comme résultat une représentation appropriée et pour qui il y aura des opérateurs génétiques sensés. Par exemple, dans le TSP si on utilise les opérateurs standards de mutation et de croisement, un AG va explorer l'espace de toutes les combinaisons des noms de villes quand, en fait, c'est l'espace de toutes les permutations qui a de l'intérêt. Le problème évident est que comme N (le nombre de villes qui constituent le tour) augmente, et vu que l'espace de permutations est un tout petit sous-ensemble de l'espace de combinaisons, la puissante heuristique d'échantillonnage de l'algorithme génétique est rendue impuissante par un mauvais choix de représentation.

Pendant les premières phases de l'IA, les solveurs des problèmes généraux (SPG) étaient dessinés comme des outils génériques pour approcher les problèmes complexes. Toutefois, comme il est arrivé, **il était nécessaire d'incorporer la connaissance spécifique du problème en raison de la complexité intraitable de ces systèmes.** .Maintenant, l'Histoire Se répète : jusqu'à récemment, les algorithmes génétiques étaient perçus comme des outils génétiques utiles pour l'optimisation de plusieurs problèmes difficiles.

Cependant, le besoin d'incorporer la connaissance spécifique du problème aux algorithmes génétiques était reconnu, depuis un certain temps, dans quelques articles des chercheurs. Il paraît que les AGs (comme SPGs) doivent, eux aussi, être indépendants du domaine pour qu'il soient utiles dans plusieurs applications. Pour cela, il n'est du tout surprenant de voir que les programmes évolutif incorporant la connaissance spécifique du problème dans la structure de données du chromosome, et les opérateurs "génétiques" spécifiques ont une meilleure performance. La différence conceptuelle essentielle entre les algorithmes génétiques et les programmes évolutifs est présentée dans les figures suivantes :

Les AG classiques qui agissent sur des chaînes binaires, demandent une modification du problème original pour le transformer en une forme appropriée (qui convient aux AG) ; ceci devrait inclure une application entre les solutions potentielles et la représentation linéaire, en faisant attention aux décodeurs ou des algorithmes de réparation, etc. Ceci n'est pas normalement une tâche facile. D'un autre côté, les PE devraient laisser le problème inchangé, mais ils modifient la représentation chromosomique de la solution potentielle (en utilisant des structures "naturelles" de données), et ils appliquent des opérateurs "génétiques" appropriés. En d'autres mots, pour résoudre un problème non-trivial en utilisant les programmes de l'évolution, on peut soit transformer le problème en une forme appropriée pour l'algorithme génétique (Fig.2.a), soit transformer l'algorithme génétique pour qu'il convienne au problème (Fig.2.b). Clairement, les AGs classiques adoptent la première forme, tandis que les PEs adoptent la deuxième.

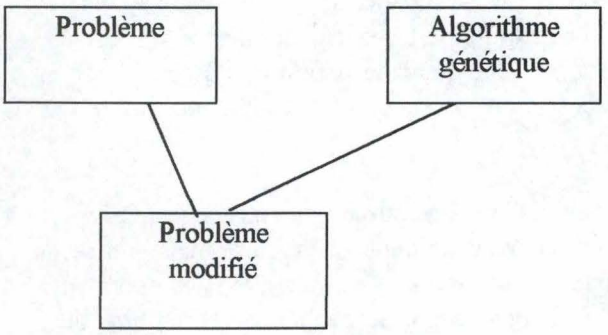


Figure 2.a.

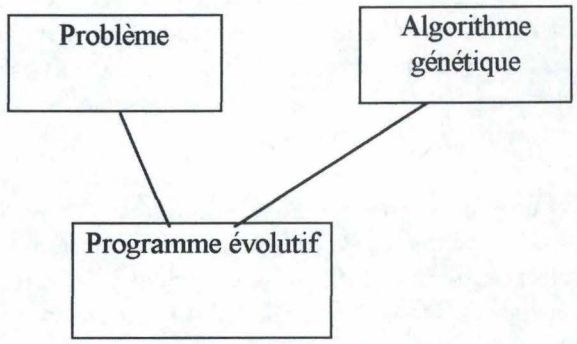


Figure 2.b.

Il est bien difficile de tracer une ligne entre les AG et les PE. Que faut-il pour un PE pour qu'il devienne un AG ? Le maintien de la population des solutions potentielles ? Le processus de sélection basé sur l'appropriation des individus ? Les opérateurs de recombinaison ? L'hypothèse des blocs de construction ? Tout ceux-ci ensemble ? Plusieurs chercheurs ont constaté et reconnu la force et la puissance de ces multiples modifications. Par exemple, un chercheur a utilisé l'hybridation entre la technique des AG et les algorithmes courants, en utilisant l'encodage de ces derniers, puis il a incorporé les points positifs de deux techniques, et enfin il a adapté les opérateurs génétiques. On voit que les AG hybrides et les PE ont une idée en commun : on part des AG classiques avec la chaîne binaire vers des systèmes plus complexes, en utilisant les structures de données appropriées et les opérateurs génétiques assortis. Ceci implique l'existence d'un ou plusieurs algorithmes courants- traditionnels- disponibles qui concernent le domaine du problème, et qui seront à la base de la construction des algorithmes hybrides.

Quels sont les points forts et les points faibles de la programmation évolutive ? Il semble que le point fort le plus important des PE est son applicabilité très large. En effet, les résultats sont souvent exceptionnels : les systèmes fonctionnent beaucoup mieux que les softwares commerciaux disponibles. Un autre point fort lié aux PE est qu'ils sont naturellement en parallèle- donc exécutables en parallèle. De l'autre côté, on doit admettre que les PE ont une base théorique faible et pauvre. Les expérimentant avec différentes structures de données et des croisements et des mutations modifiés exige une analyse prudente qui peut garantir une performance raisonnable. Cependant, certains PE possèdent des fondations théoriques : En ce qui concerne les stratégies évolutives appliquées aux problèmes réguliers, une propriété de convergence peut être démontrée. Les AG ont le « théorème du schéma » qui explique pourquoi ils fonctionnent, malgré que ce théorème a reçu dernièrement beaucoup de critiques. Pour les autres PE on a seulement des résultats intéressants.

En général, les stratégies de solutions des problèmes en IA sont divisées en deux catégories : des méthodes "forts" et des méthodes "faibles". Une méthode faible fait plusieurs suppositions concernant le

domaine du problème. Pour cela ils possèdent habituellement une applicabilité large. D'un autre côté, ils peuvent souffrir de l'explosion des coûts des solutions combinatoires quand on passe aux problèmes de tailles plus grands. Celle-ci peut être évitée en faisant des suppositions fortes sur le domaine du problème, et par conséquent, on peut exploiter ces suppositions dans les méthodes qu'on utilise pour résoudre le problème. Mais un inconvénient de ces méthodes fortes est leur applicabilité limitée : le plus souvent ils demandent une reconception significative même quand ils sont appliquées à des problèmes similaires.

Les PE se situent quelque part entre les méthodes fortes et les méthodes faibles. Certains programmes évolutifs (comme les programmes génétiques) sont bien faibles si on ne fait pas aucune supposition sur le domaine du problème. Quelques autres sont plus spécifiques avec un degré de dépendances variables. C'est un peu ironique : les AG sont considérés comme des méthodes faibles ; cependant, en présence des contraintes non-triviales, ils deviennent rapidement des méthodes fortes.

Tous ces recherches ont mené à la création d'une nouvelle méthodologie de programmation qu'on appelle parfois EVA (EVolution progrAmming). Dans un tel environnement, un programmeur choisirait pour un certain problème les structures de données avec les opérateurs génétiques appropriés, ainsi que la fonction d'évaluation et l'initialisation de la population (les autres paramètres sont "tunés" par un autre processus génétique). Un point important à remarquer dans cette méthodologie, c'est l'implémentation des programmes pour contrôler l'évolution du processus évolutif qui se déroule grâce au "moteur de l'évolution"- le moteur de l'évolution est représenté par une "société de microprocesseurs" .

La motivation principale du nouveau environnement de programmation est de fournir des outils de programmation basés sur une structure parallèle ; ce qui est très important car le parallélisme va sûrement changer notre façon de penser et d'utiliser l'ordinateur. Il promet de mettre à notre portée des solutions pour des problèmes et des frontières à notre connaissance qu'on a jamais rêvé à avoir auparavant. La riche variété d'architectures va mener à la découverte des nouvelles et plus efficaces solutions pour les anciens et nouveaux problèmes en même temps .

Enfin, une approche très intéressante de la méthodologie de la programmation évolutive a été développée récemment par Koza. Au lieu de construire un programme évolutif, Koza suggère que le programme désiré doit lui-même évoluer durant le processus d'évolution. Autrement dit, au lieu de résoudre un problème, et au lieu de construire un programme pour le résoudre, on va plutôt chercher dans l'espace de programmes d'ordinateur possibles le meilleur (le plus approprié). Koza a développé une nouvelle méthodologie nommé Programmation Génétique, qui fournit un moyen pour exécuter cette recherche. Une population des programmes d'ordinateur est créée, et les programmes individuels se font la compétition entre eux où les programmes les plus faibles meurent, et les programmes forts se reproduisent (par croisement et mutation)...

Il existe cinq étapes majeures à suivre pendant l'utilisation de la programmation génétique pour un problème particulier. Ce sont :

- Sélection des terminaux
- Sélection d'une fonction
- Identification de la fonction d'évaluation
- Sélection des paramètres du système
- Sélection de la condition de terminaison

Il est important de noter que la structure qui subit l'évolution est un programme d'ordinateur structuré et hiérarchique. L'espace de recherche est un hyperespace de programmes valides, qui peut être vu comme un espace d'arbres enracinés. Chaque arbre est composé de fonctions et de terminaux appropriés au domaine du problème particulier ; l'ensemble de tous les terminaux et les fonctions sont choisis a

priori d'une façon que quelques arbres composés donnent une solution. La population initiale est composée de ce genre d'arbres; La construction d'un arbre aléatoire est simple. La fonction d'évaluation assigne une valeur d'appropriation qui évalue la performance d'un arbre (programme). L'évaluation est basée sur un ensemble de cas de test présélectionnés ; en général, la fonction d'évaluation retourne- renvoie- la somme de distances entre les résultats corrects et les résultats obtenus dans tous les cas de test.

La sélection est proportionnelle telle que chaque arbre a une probabilité d'être sélectionné à la nouvelle génération proportionnelle à son appropriation- son aptitude-. L'opérateur primaire est un croisement qui produit deux descendants à partir de deux parents sélectionnés. Le croisement crée la descendance en échangeant des sous-arbres entre deux parents. Il y a aussi des autres opérateurs de mutation, de permutation, d'édition et de définition de l'opération de construction des blocs.

Il existe une différence significative entre l'environnement de la programmation évolutive EVA et le paradigme de la programmation génétique de Koza. Néanmoins, tous les deux sont basés sur les algorithmes génétiques et le principe de l'évolution, en toute généralité. Il est intéressant de noter qu'il y a plusieurs autres approches d'apprentissage, d'optimisation et de résolution des problèmes, qui sont basées sur des autres métaphores de la nature- parmi les exemples les plus connus on trouve les réseaux neuronaux et le recuit simulé.

Pour finir, on peut constater qu'il y a un intérêt grandissant pour tous ces domaines ; l'orientation la plus fertile et la plus "challenging" est apparemment la "recombinaison" de certaines de ces idées qui sont pour le moment éparpillés dans des différents domaines.

Chapitre 2 : Les Algorithmes Génétiques

2.1. Philosophie des AG

Les algorithmes génétiques sont des algorithmes d'exploration fondés sur les mécanismes de la sélection naturelle et de la génétique. Ils utilisent à la fois les principes de la survie des structures les mieux adaptées; et les échanges d'information pseudo-aléatoire, pour former un algorithme d'exploration qui possède certaines des caractéristique de l'exploration humaine. A chaque génération, un nouvel ensemble de créatures artificielles (des chaînes de caractères) est créé en utilisant des parties des meilleurs éléments de la génération précédente; ainsi que des parties innovatrices, à l'occasion. Bien qu'utilisant le hasard, **les AG ne sont pas purement aléatoires.** Ils exploitent efficacement l'information obtenue précédemment pour spéculer sur la position de nouveaux points à explorer, avec l'espoir d'améliorer la performance.

En effet; les algorithmes génétiques explorent des espaces de solution beaucoup plus vastes que les programmes classiques. En outre, l'étude des effets de la sélection naturelle sur les programmes ; dans des conditions contrôlées et bien comprises; pourrait montrer comment la vie et l'intelligence ont évolué sur la Terre. La plupart des organismes évoluent par deux mécanismes: la sélection naturelle et la reproduction sexuée.

La sélection naturelle détermine quels membres d'une population survivent et se reproduisent; la reproduction sexuée assure le brassage et la recombinaison des gènes nouvelles. Lorsqu'un spermatozoïde et un ovule fusionnent; leurs chromosomes s'apparient et s'échangent des segments; Ce mélange permet une évolution beaucoup plus rapide que si les individus recevaient tous leurs gènes d'un même parent et que si les mutations étaient les seules causes de variations génétiques (bien que les organismes unicellulaires ne s'accouplent pas comme les animaux, ils échangent également du matériel génétique, et leur évolution peut se décrire de la même façon).

La sélection est simple: un organisme meurt s'il n'est pas adapté ; c'est-à-dire s'il n'échappe pas à ses prédateurs ou s'il ne trouve pas de proies. De même, les informaticiens savent reconnaître et éliminer les algorithmes médiocres : pour savoir si un programme classe correctement des nombres par ordre croissant, par exemple, on examine si chaque nombre de la liste produite par le programme est supérieur au nombre précédent de la liste.

Depuis des millénaires, nos ancêtres utilisent la reproduction et la sélection pour améliorer les récoltes, les chevaux ou les roses. Comment agir de même avec les programmes informatiques ? Le problème principal est la construction d'un "code génétique" qui représente la structure des programmes, tout comme le code génétique de l'ADN représente la structure des êtres humains ou des souris : dans la plupart des cas, une mutation d'un programme rédigé en langage informatique FORTRAN par exemple, ou le mélange de deux programme écrits dans le même langage engendrent des programmes incohérents, que les ordinateurs ne peuvent exécuter.

Quand on a voulu mêler l'informatique et la biologie de l'Evolution, à la fin des années 1950, les résultats ont été médiocres par ce que, fondés sur les connaisssances biologique de l'époque, ils privilégiaient le mécanisme de mutation- plutôt au celui de reproduction- pour former de nouvelles combinaisons de gènes. Puis, au début des années 1960, à l'Université de Berkeley, Hans Bremermann imagina une sorte de reproduction, où les caractéristique des descendants étaient déterminées par addition des gènes correspondants chez les deux parents. Cette reproduction ne pouvait pas toutefois avoir lieu que lorsque la somme des caractéristiques avait un sens.

John Holland qui a étudié mathématiquement les mécanismes d'adaptation, lui a semblé alors que la recombinaison des gènes, lors de la reproduction, était cruciale pour l'évolution. Au milieu des années 1960 ; il a trouvé - ou plus exactement a inventé - une technique de programmation, l'algorithmique génétique ; qui permet l'évolution des programmes par reproduction et par mutation. Dans les dix années qui ont suivi, J.Holland a développé cette technique en créant un code génétique qui pourrait représenter la structure de tout programme informatique.

J.Holland, ainsi que ses collègues et étudiants ont continué par après à développer les algorithmes génétiques. Leurs recherches avaient deux objectifs principaux : (1) mettre en évidence et expliquer rigoureusement les processus d'adaptation des systèmes naturels, et (2) concevoir des systèmes artificiels (en l'occurrence des logiciels) qui possèdent les propriétés importantes des systèmes naturels. Cette approche a débouché sur les découvertes importantes à la fois dans les sciences des systèmes naturels et dans celles des systèmes artificiels.

La recherche sur les algorithmes génétiques a pour souci principal l'amélioration de la robustesse, l'équilibre entre la performance et le coût nécessaire à la survie dans des environnements nombreux et différents.

La robustesse des systèmes artificiels est capitale à plus d'un titre. Si les systèmes artificiels peuvent être rendus plus robustes, les modifications coûteuses peuvent être réduites ou même éliminées. Si des niveaux d'adaptation supérieurs peuvent être atteints, les systèmes existants peuvent remplir leur fonction mieux et plus longtemps. Les concepteurs de systèmes artificiels, aussi bien logiciels que matériels, dans les domaines de l'ingénierie traditionnelle, de l'informatique, ou de l'entreprise, ne peuvent que s'émerveiller devant la robustesse, l'efficacité, et la flexibilité des systèmes biologiques. Des caractéristiques telles que l'auto-réparation, l'autoguidage, et la reproduction sont la règle dans les systèmes biologiques, tandis qu'elles existent à peine dans les systèmes artificiels les plus sophistiqués. Ainsi, nous sommes amenés à cette intéressante conclusion : quand une performance robuste est souhaitée (quand ne l'est-elle pas?), la nature fait mieux que les ingénieurs ; c'est donc à travers l'étude minutieuse de l'exemple biologique que l'on peut le mieux apprendre les secrets de l'adaptation et de la survie. Cependant; l'argument de la "beauté naturelle" n'est pas le seul à nous convaincre de l'intérêt des algorithmes génétiques. Il a été prouvé théoriquement et expérimentalement que les algorithmes génétiques sont des procédures **robustes** d'exploration d'espaces complexes. L'ouvrage originel sur ce thème est celui de Holland (1975) *"Adaptation in Natural and Artificial Systems."* Un grand nombre d'articles et de thèses de doctorat établissent la validité de la technique pour les applications d'optimisation de fonctions et de contrôle.

Ayant été reconnus comme une approche valide des problèmes nécessitant une exploration performante et économique du point de vue du calcul, les AG sont maintenant appliqués plus largement ; aux domaines des affaires; à la recherche scientifique en général, ainsi que pour l'ingénierie. Les raisons de ce nombre d'un point de vue de calcul, mais sont cependant très performants dans leur recherche d'amélioration. De plus, ils ne sont pas fondamentalement limités par des hypothèses contraignantes sur le domaine d'exploration (hypothèses concernant la continuité, l'existence de dérivées, l'unimodalité, entre autres).

En effet, la recherche de solution dans un espace complexe implique souvent un compromis entre deux objectifs apparemment contradictoires: l'exploitation des meilleures solutions disponibles à un moment donné et une exploration robuste de l'espace des solutions possibles. Les méthodes de type "grimpeurs" ("hill-climbing" en anglais) représentent le paradigme de l'exploitation au détriment de la non-globalité de l'optimum trouvé, sauf dans des cas bien particuliers (le "grimpeur" procède itérativement en tentant à chaque pas de trouver localement une solution intermédiaire meilleure que la solution courante, tout comme l'alpiniste qui cherche à rejoindre un sommet) ; les méthodes de type "recherche aléatoire" représentent l'autre extrémité mais sont souvent inacceptables en pratique à cause de leur lenteur

excessive et de leur ignorance aveugle. Les Algorithmes Génétique sont une Classe de stratégies de recherche réalisant - comme on l'a déjà cité - un compromis équilibré et raisonnable entre l'exploration et l'exploitation ; en effet des analyses théoriques ont montré que les AG géraient ce compromis de façon presque optimale.

Leur fonctionnement repose sur une heuristique très simple : les meilleures solutions seront trouvées dans des régions de l'espace de recherche contenant des proportions relativement élevées de bonnes solutions et, en outre, ces régions peuvent être identifiées par un échantillonnage robuste et judicieux de l'espace des solutions.

Ainsi les AG risquent moins d'être piégés dans des minima locaux que les algorithmes classiques, parce qu'ils explorent en parallèle un ensemble de solutions aux problèmes posés. En théorie, bien que ces solutions soient améliorées de génération en génération, on n'a aucune garantie que l'AG découvre la solution optimale. En pratique cependant, il s'approche rapidement assez près de la solution optimale pour être non seulement souvent utile mais encore, parfois, irremplaçable.

2.2 Un aperçu général

Les mécanismes d'un algorithme génétique de base (simple Genetic Algorithm ou SGA) sont étonnamment simples, et ne mettent en jeu rien de plus compliqué que des copies de chaînes et des échanges de morceaux de chaînes. Les raisons pour lesquelles ce procédé simple fonctionne sont bien plus complexes et subtiles. La simplicité de mise en oeuvre et l'efficacité constituent deux des caractéristique les plus attrayantes de l'approche proposée par les algorithmes génétiques.

Commencons d'abord par décrire schématiquement le SAG (Fig. 2.1.).

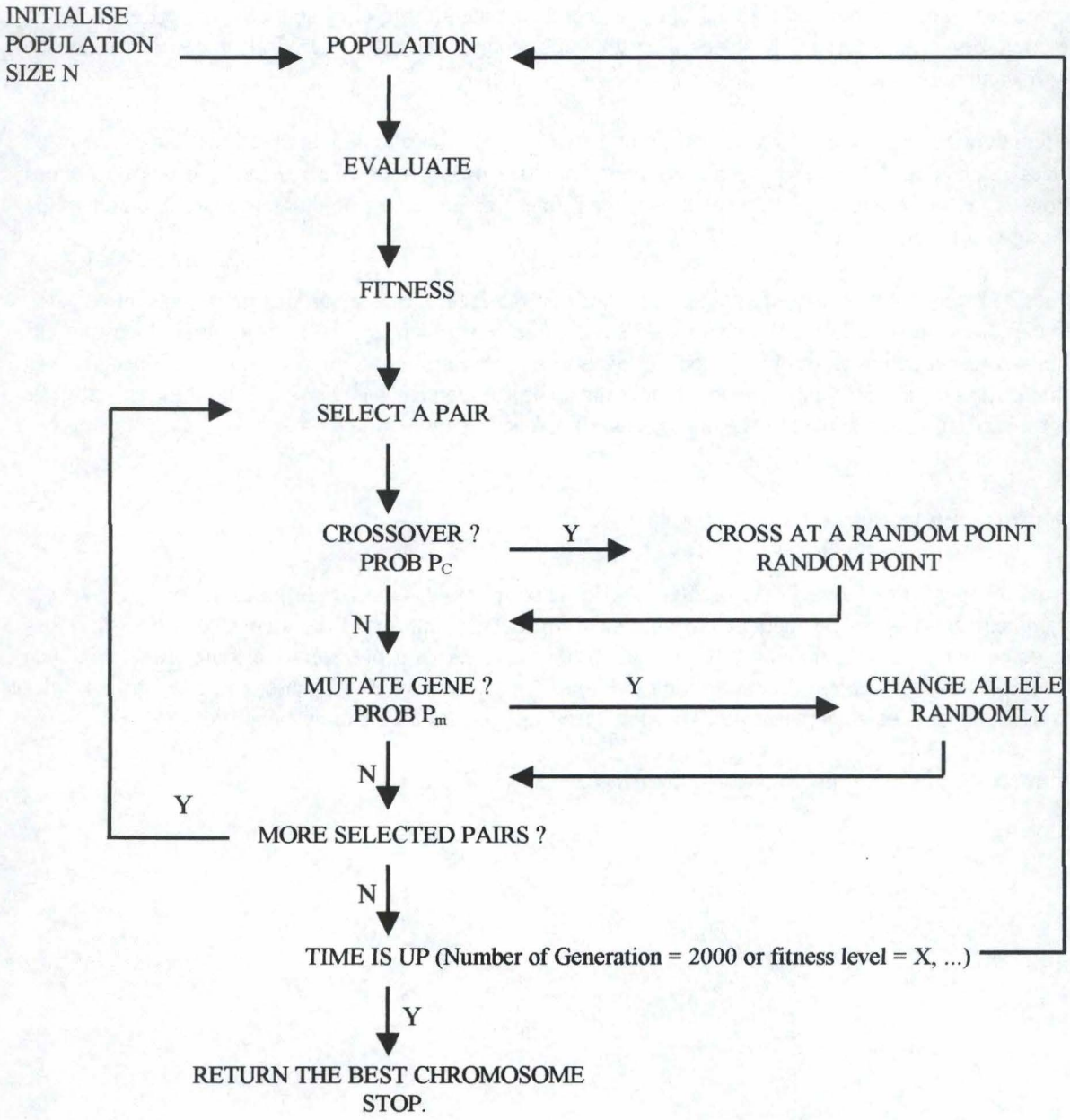


Figure 2.1 The Simple Genetic Algorithm (SGA)

Avant de décrire plus précisément cet algorithme, quelques mots d'explication concernant l'organigramme de la figure 2.1. qu'on peut l'expliquer en 6 étapes:

1. Initialiser - générer aléatoirement ou heuristiquement - une population de Chromosomes
2. Evaluer chaque chromosome de la population selon un certain critère de fitness-d'aptitude-
3. Créer des nouveaux chromosome par la reproduction - l'accouplement - des chromosomes actuels; Appliquer la recombinaison - crossover - et la mutation autant des opérations de reproduction.
4. Supprimer des membres de la population - en nombre égal au nombre de nouveaux chromosomes pour garder le nombre total d'individus constant - pour faire place aux nouveaux chromosomes
5. Evaluer les nouveaux Chromosomes, puis insérer les dans la population

6. Si le temps - ou le nombre de générations, ou le niveau de fitness - est atteint Alors Arrêter et Retourner - Renvoyer - le meilleur Chromosome ; sinon, aller à l'étape 3.

Décrivons maintenant un peu plus précisément un Algorithme Génétique Simple (SGA) : il s'agit d'un algorithme itératif de recherche globale dont le but est d'optimiser une fonction définie par l'utilisateur (le critère, la fonction de coût ou la fonction de profit), appelée **fonction d'adéquation** - ou d'aptitude - (traduction du mot anglais **fitness**) ; pour atteindre cet objectif, l'algorithme travaille en parallèle - en **data parallelism**[1], au moins en théorie, et pour cette raison les AG se prêtent très bien aux machines massivement parallèles - sur une population de points candidats, appelés **individus** ou **chromosomes,** distribués dans l'entièreté de l'espace de recherche. La fonction d'adéquation d'un individu est normalement indépendante des autres individus de la population (elle est explicitement donnée par l'utilisateur) ; on pourrait concevoir une fonction d'adéquation implicite qui dépendrait du reste de la population via certaines interactions entre individus; on obtiendrait alors la **co-évolution** d'individus, mais nous ne nous étendrons pas sur ce concept, malgré son énorme richesse.

Chaque individu ou chromosome est constitué d'un ensemble d'éléments appelés **caractéristiques** ou **gènes**, pouvant prendre plusieurs valeurs **(allèles)** appartenant à un alphabet non nécessairement numérique. Dans l'algorithme de base, les allèles possibles sont 0 et 1, et un chromosome est donc une chaîne binaire. Le but est donc de chercher la combinaison optimale de ces éléments, qui donne lieu au maximum d'adéquation.
A chaque itération, appelée **génération**, est créée une nouvelle population avec le même nombre d'individus. Cette nouvelle génération consiste généralement en des individus mieux "adaptés" à l'environnement tel qu'il est représenté par la fonction d'adéquation. Au fur et à mesure des générations, les individus vont tendre en général vers l'optimum de la fonction d'adéquation. La génération d'une nouvelle population à partir de la précédente s'effectue en trois étapes :

a) **évaluation** : l'Algorithme Génétique évalue la fonction d'adéquation f_i de chaque individu i de l'ancienne population ;

b) **sélection** : l'Algorithme Génétique sélectionne les individus sur la base de leur fonction d'adéquation; plus précisément; l'opérateur de sélection sélectionne chaque individu i avec une probabilité $f_i / \Sigma f_i$ (la somme étant prise sur les N individus constituant la population); comme l'opérateur de sélection est appliqué n fois, l'espérance mathématique du nombre d'enfants de i sera $n f_i / \Sigma f_i$. Les individus sélectionnés constituent une population intermédiaire P'

c) **reproduction avec croisement et mutation:** l'AG recombine les individus sélectionnés au moyen d'opérateurs génétiques tels que la mutation et le croisement , qui, d'un point de vue algorithmique, peuvent être considérés comme des mécanismes servant à changer localement les solutions représentées par les parents (mutation) ou à les combiner (croisement). La mutation agit en modifiant aléatoirement un ou plusieurs gènes d'un chromosome, tandis que le croisement échange certains gènes d'un parent avec ceux de l'autre. Ces deux opérateurs sont appliqués avec des probabilités fixées ($\mathbf{P_c}$ pour le croisement et $\mathbf{P_m}$ pour la mutation) sur chacun des individus de la population intermédiaire P' Plus précisément, pour le croisement, on choisira au hasard 2 parents dans la population P'et un site de croisement (un nombre de 1 à *l*, si *l* est la longueur du chromosome) ; soit k, l'indice de site; on obtiendra alors 2 enfants en prenant d'une part les k premiers gènes du premier parent et les (*l-k*) derniers gènes du second parent, et pour l'autre enfant -car un croisement donne toujours, au moins en SAG, 2 descendants - les *k* premiers gènes du second et les (*l-k*) derniers gènes du premier parent. Cette méthode, appelée croisement à 1 point, n'est pas unique ; il existe aussi des croisements multipoints et le croisement uniforme qui consiste à choisir avec la même probabilité un bit de l'un ou l'autre parent pour chaque site de l'enfant. On recommence le croisement avec les autres parents, jusqu'à obtenir une nouvelle population P" qui contient toujours n individus - car à

[1] C'est quand on exécute la même opération sur plusieurs items - ou données - en même temps, ce qui donne en pratique une très grande économie d'échelle en calcul - computing-.

chaque croisement les deux descendants ne remplacent pas leurs parents mais deux chaînes mal adaptées, et ainsi le nombre total d'individus N reste constant-.

Enfin, chaque gène des individus de la population P" peut subir une mutation avec une probabilité $\mathbf{P_m}$; lorsqu'un gène est muté, sa valeur est remplacée par une autre appartenant au même alphabet. Après mutation, les individus de P" constitueront la nouvelle génération. Et le cycle continue : évaluation, sélection ; reproduction, évaluation, etc. Il est clair que l'opérateur de croisement a un grand rôle dans la recherche globale de l'espace par combinaison de "blocs de construction" ; l'opérateur de mutation a un rôle théoriquement plus marginal: il est là pour éviter une perte irréparable de la diversité (supposons par exemple, que tous les individus d'une population commencent par la même valeur du premier gène : cette valeur ne pourrait pas changer s'il n'y avait pas de mutation).

Il y a donc trois paramètres de base pour le fonctionnement d'un AG: le nombre d'individus dans la population (**n**), les taux de mutation $\mathbf{P_m}$ et de croisement $\mathbf{P_c}$. Trouver de bonnes valeurs à ces paramètres est un problème parfois délicat. La valeur de n dépend fort du problème (en particulier de la longueur l du chromosome), tandis que des valeurs de bonne pratique pour $\mathbf{P_m}$ (0.005) et $\mathbf{P_c}$ (0.7) sont souvent utilisées.

Pour mieux faire comprendre le principe, imaginons le scénario suivant: une population initiale est composée de 4 individus. Chaque individu est caractérisé par 3 traits: ses capacités de jardinier, sa qualité de cuisinier et ses envies cleptomanes. Chaque trait peut prendre 2 valeurs (0 ou 1), suivant que l'individu ait ou n'ait pas la caractéristique. L'environnement est modélise par une fonction d'adéquation définie pour chacune des combinaisons des 3 caractéristiques ; par exemple l'individu (0;0;0), qui n'est ni cuisinier, ni jardinier, ni cleptomane, aura une qualité d'adéquation nulle, ce qui traduit le fait que, ne pouvant se fournir sa nourriture, il dépérira avant d'avoir pu se produire (ou n'aura tout simplement pas assez de force pour le faire); l'individu (0;1;0) ne sera pas mieux loti car, malgré ses talents de cuisinier, il n'aura rien à se mettre sous la dent; l'individu (1;0;0) aura néanmoins ses chances en se nourrissant de légumes crus. (0;0;1) survit en volant, tandis que (0;1;1) arrive à améliorer la qualité de ses vols en les cuisinant; (1;0;1) combine ses vols avec les produits de son jardin et (1;1;0) est l'individu idéal. Quant à (1;1;1), il perd du temps et de l'énergie à voler pour son propre plaisir, ce qui donne une capacité d'adéquation inférieure à celle de (1;1;0) ces considérations sont résumées dans le tableau suivant, en se rappelant que l'ordre est: (jardinier, cuisinier, cleptomane):

individus	qualité d'adéquation
(0;0;0)	1
(0;0;1)	3
(0;1;0)	1
(0;1;1)	4
(1;0;0)	2
(1;0;1)	4
(1;1;0)	5
(1;1;1)	4

Supposons la population initiale - qui est générée aléatoirement - constituée des individus {(1;0;0), (0;1;0), (0;0;1), (1;1;1)}. Voyons comment agit un AG pour trouver la solutions optimale :

Première Phase de l'AG: **évaluation** des qualités d'adéquation (1;0;0) se voit attribué un score 2, (0;1;0) a un score de 1, (0;0;1) a 3, tandis que (1;1;1) a 4.

Deuxième Phase: **sélection**
L'algorithme attribue à chaque individu une chance de participer à la phase de reproduction proportionnelle au rapport de sa qualité d'adéquation par rapport à la qualité moyenne. Ainsi, on

attribuera à l'individu 1 la probabilité de 2/ (2+1+3+4), soit 2 chances sur 10. Cela entraîne aussi qu'il aura en moyenne (2/10) * 4 enfants (espérance mathématique) dans la génération suivante. On procède alors au tirage au sort- celui-ci, avec les $\mathbf{P_m}$ et $\mathbf{P_c}$ donnent aux AG le caractère des processus probabilistes ou stochastiques -, suivant le principe de "la roue de la Fortune": 2/10 de la roue sont attribués à l'individu 1,1/10 à l'individu 2, 3/10 à l'individu 3, et 4/10 à l'individu 4. La roue est lancée autant de fois qu'il y a d'individus dans la population . Supposons que le tirage ait donné: l'individu 4 (2 fois), l'individu 3 (1 fois) et l'individu 1 (1 fois). Nous sommes maintenant en présence d'une population intermédiaire : {(1;1;1),(1;1;1),(0;0;1),(1;0;0)},respectivement appelés {1bis, 2bis, 3bis, 4bis}

Troisième phase : **reproduction**
Au cours de cette phase, les couples sont choisis au hasard. Supposons que 1 bis "aille avec" 4bis et 2bis avec 3bis. Des sites de croisement sont ensuite choisis au hasard:

1 1 | 1 et 1 | 1 1
1 0 | 0 et 0 | 0 1

ce qui donne:

1 1 0 et 1 01
1 0 1 0 11

Les 4 nouveaux membres sont donc : {(1;1;0), (1;0;1), (1;0;1), (0;1;1)}. Ils subissent alors des mutations (en général la probabilité de mutation est très faible) supposons que seule la caractéristique 2 du 3^{ième} individu soit mutée. Finalement, la nouvelle population formant la deuxième génération, sera constituée des 4 individus :{(1;1;0),(1;0;1),(1;1;1),(0;1;1)}.On remarquera que l'individu idéal y est maintenant présent.

Ensuite, en recommençant interactivement les 3 phases ci-dessus, on tendra vers une population constituée presque exclusivement de (1;1;0) (qui a une valeur de fonction d'adéquation supérieure aux autres) ; il y a aura de là cependant toujours quelques mutants pour préserver la diversité de la population et donc éventuellement réagir à un changement quelconque de l'environnement.

- **Quelques remarques importantes :**

1. Vu que les AG sont des processus itératifs stochastiques, **on ne peut pas garantir leur convergence** ; la condition d'arrêt (de terminaison) peut être spécifiée comme un certain nombre (fixé à l'avance), de générations, ou un seuil minimal d'adéquation - de fitness - à atteindre, etc.
2. Il peut arriver, parfois, que dans ma nouvelle population - produite par sélection, croisement et mutation - le meilleur chromosome (ou la meilleure solution) qui était présent à l'itération précédente, disparaît (sort) de cette nouvelle génération ; cela ne pause aucun problème car **la valeur moyenne d'adéquation de la population en sa totalité augmente continuellement**- c'est une fonction croissante avec parfois des paliers-. De ce fait il y a des implémentations des AG, et pour des raisons d'optimisation on garde une trace du " meilleur chromosome à jamais "- qu'on a jusqu'à maintenant eu - dans une location à part, et de cette façon on reporte cette meilleure valeur durant la totalité du processus.
3. **Dans les systèmes du traitement en temps réel, l'utilisations des AG crée des problèmes sérieux vu l'imprévisibilité- unpredictability- du temps dans lequel les AG vont atteindre la solution.** Ce problème du temps de réponse vient, comme on l'a déjà vu en point 1 ci-dessus du fait que les AG sont des processus discrets, stochastique et non-linéaires. Ces caractéristiques des AG les rendent aussi difficiles à réagir promptement à un environnement changeant et variable.

4. **L'implémentation d'un algorithme génétique - ou d'un algorithme évolutif en général- peut être dans certain cas assez coûteux- en temps bien sûr-** , car les populations des solutions impliquées dans le processus sont peut-être liées à des calculs intensifs (lourds) des fonctions d'adéquation de ces solutions- chromosomes -. **Une solution à ce problème est de "paralléliser" le processus**, à condition d'apporter quelques modifications judicieuses aux algorithmes existants.

2.2.1. Des variantes de l'Algorithme Génétique Simple (SGA)

Nous venons de décrire l'Algorithme Génétique de base, mais il existe bien d'autres variantes aux AG. En fait, nous avons remarqué que chaque utilisateur concevait en général son propre AG suivant son application propre ; il n'y a là rien de répréhensible : les AG de base, sur lesquels, il est vrai, une grande partie de la théorie a été établie, ont des côtés peu pratiques et trop aveugles parce qu'on a voulu conserver leur indépendance par rapport aux problèmes ; il est donc normal que chacun y ajoute quelques heuristiques qui en améliorent l'efficacité ; le tout est de conserver les principes de base. Nous citerons brièvement des variantes désormais classiques :

1. Le mécanisme de sélection proportionnelle a ses avantages, mais aussi ses inconvénients, en particulier, si un individu est très supérieur à la moyenne, il constituera presque exclusivement la population suivante et on aura perdu toute diversité (risque de convergence prématurée) ; de même, s'il y a très peu de différence entre les qualités d'adéquation des différents individus, la recherche stagnera et se comportera plutôt comme une "promenade" aléatoire (« random walk »). Il faut donc trouver une solution de compromis pour éviter ce genre d'excès. Une solution possible procède par un changement d'échelle de la fonction d'adéquation : on s'arrange pour que la fonction d'adéquation minimum et la fonction d'adéquation maximum aient des valeurs données en adoptant un changement d'échelle statique ou dynamique , linéaire ou exponentiel ; évidement, le changement d'échelle est indispensable quand les valeurs de la fonction d'adéquation sont négatives puisque la sélection consiste à ranger les individus par ordre décroissant de leur fonction d'adéquation (par ordre croissant si l'on a un problème de minimisation) et à attribuer une probabilité de sélection P_i selon le domaine de valeur :
$$n * P_i = \varnothing - (r_i - 1) * (2\varnothing - 2) / (n-1)$$ (= nombre moyen d'enfants de l'individu i)
où r_i est le rang de l'individu i, n le nombre d'individus et $\varnothing$ la pression de sélection ($\varnothing \in [1,2]$ et représente le nombre moyen d'enfants du meilleur individu ,le moins bon en ayant nécessairement $(2-\varnothing)$. Cette stratégie de sélection ,appelée sélection par rangement, évite les inconvénients de la sélection proportionnelle et représente en fait un changement d'échelle dynamique non linéaire très particulier.
2. On peut décider qu'à chaque génération, on ne renouvellera qu'une partie restreinte de la population suivant un taux de renouvellement τ. Dans ce cas on choisit τ_n individus de la population que l'on recombine et mute suivant les techniques classiques : les enfants produits peuvent remplacer leurs parents dans la nouvelle population, ou encore remplacer les individus de la population qui leur sont le plus proches génétiquement ("crowding scheme").
3. Il peut être intéressant de garder intact le meilleur individu de la population lors du passage d'une génération à la suivante : la **stratégie élitiste** concrétise cette idée en remplaçant le meilleur individu d'une population dans la population suivante et ce, sans aucune altération (mutation) ; cet individu reste toutefois candidat pour la phase de reproduction habituelle. Cette stratégie, simple mais efficace ; a été adaptée dans beaucoup de travaux.
4. Quelquefois, l'utilisateur est intéressé à trouver non pas l'optimum global du problème, mais l'ensemble des optima ; pour cela, on peut introduire un mécanisme de partage ("sharing scheme") basé sur l'idée de niche écologique; les individus occupant une même niche doivent se partager les ressources disponibles pour celle-ci, ce qu'on traduit par une modification de la fonction d'adéquation :
$$f'_i = f_i / \Sigma s(d(i,j)),$$

où f' est la nouvelle fonction d'adéquation, d(i,j) une mesure de la distance entre 2 individus i et j (la somme est effectuée sur tous les individus j de la population) et s(d) la fonction de partage, qui est décroissante par rapport à la distance d, vaut 1 pour une distance nulle et tend vers o si la distance est très grande. Avec cette nouvelle fonction d'adéquation, on pourra observer la formation de sous-populations au niveau des optima locaux, avec d'autant moins d'individus que l'optimum local est peu élevé.

De nombreuses versions modifiées des AG originaux ne se basent plus sur un codage binaire des paramètres à optimiser, mais travaillent directement sur les paramètres eux-mêmes. Ces versions des Algorithmes Génétiques, que nous appellerons **Algorithmes Génétiques codés-réels,** offrent généralement l'avantage d'être mieux adaptées aux problèmes d'optimisation numérique continue, d'accélérer la recherche et de rendre plus aisé le développement de méthodes hybrides avec des méthodes plus classiques.

Ce faisant, ces versions sont sans doute plus proches des besoins et des habitudes des praticiens industriels pour la résolution de problèmes réels ; ce sont les algorithmes Génétiques originaux. Il existe une classe particulière d'AG codés réels : ce sont les Stratégies d'Evolution, qui ont la particularité d'introduire dans le chromosome des gènes supplémentaires représentant les amplitudes des mutations, si bien que ces dernières grandeurs sont elles-aussi optimisées dans le processus de recherche. Cependant, les AG codés-réels nécessitent le développement d'opérateurs spéciaux (mutations, croisements, etc..), faits sur mesure en fonction de l'application. En outre, jusqu'à présent, ces approches manquent de fondements mathématiques et ne disposent pas de la majorité des théorèmes classiques des AG, établis pour des alphabets discrets.

Un autre domaine d'étude actif est la recherche d'opérateurs génétiques différents des opérateurs de mutation, de crossing-over et d'inversion classiques. Ainsi pour résoudre un problème d'ordonnancement optimal, tel que celui du voyageur de commerce (TSP), on peut coder les solutions possibles par des chaînes de caractères telles que **ADCFGBE**, où les lettres représentent les villes dans l'ordre où elles sont parcourues: la ville **A**, puis la ville **D**, puis la ville **C**, etc.. Cet exemple montre notamment que les algorithmes génétiques ne sont pas nécessairement confinés à l'usage d'alphabets binaires. Avec le crossing-over décrit auparavant, la recombinaison de deux solutions est difficile : la chaîne **ADCFGBE** croisée avec la chaîne **ADEFGCB**, par exemple, donne naissance à la chaîne **ADCFGCB** si un crossing-over classique est effectuée à droite du troisième caractère de la chaîne. Or la solution ainsi codée n'est pas viable, puisqu'elle conduit à repasser deux fois par la ville **C** et à éviter la ville **E**. Pour ce problème et pour de nombreux autres, il convient d'adapter l'opérateur de crossing-over classique de façon qu'il n'engendre que des solutions viables. On comprend ainsi que l'usage des AG pose un double problème de choix : lors du codage des solutions recherchées, d'une part, pour les opérateurs génétique avec lesquels on explore l'espace des solutions correspondantes, d'autre part. Il est honnête de dire que, dans l'état actuel de la théorie des AG, ces choix relèvent de l'art plus que de la science.

2.3. Opérateurs des algorithmes génétiques

Un algorithme simple (SGA) composé des opérateurs suivant conduit généralement à des bons résultats pour un grand nombre d'applications:

1. Reproduction
2. Crossover
3. Mutation

Dans les paragraphes suivants, nous examinerons - sans entrer trop dans les détails - ces différents opérateurs.

2.3.1 Codages

Coder un chromosome est un des problèmes les plus importants qu'on rencontre quand on commence à résoudre un problème en utilisant les AG. Il est important car le codage peut influencer fortement les performances de l'AG qu'on utilise, et de ce fait on dit que le codage dépend beaucoup du problème. Ici on va introduire quelques codages qui sont déjà utilisés et avec succès.

- **Le codage binaire**

Le codage binaire est le plus utilisé, principalement parce que les premiers travaux sur les AG ont utilisé ce type de codage. En codage binaire chaque chromosome est une chaîne de **bits, 0** ou **1**, par exemple :

Chromosome A 1011001110000101
Chromosome B 1111100110000111

Le codage binaire génère beaucoup de chromosomes possibles même avec un petit nombre d'allèles. D'un autre côté ce codage est souvent pas naturel pour beaucoup de problèmes et parfois des corrections doivent être apportées après le crossover et/ou la mutation.

Exemple du problème : le problème du sac à dos (Knapsack).

Le problème : il y a des items avec des valeurs et des volumes donnés. Le sac à dos a une certaine capacité. Comment choisir les items de sorte qu'on maximise la valeur des items contenus dans le sac, en respectant sa capacité.

Le codage : chaque bit informe si l'item correspondant est dans le sac.

- **Le codage à permutation**

Le codage à permutation peut être utilisé dans les problèmes d'ordonnancement, comme le TSP ou l'ordonnancement des tâches. En codage à permutation, chaque chromosome est une chaîne de nombres qui représente une séquence, par exemple :

Chromosome A 153264798
Chromosome B 856723149

Ce type de codage est utile seulement pour les problèmes d'ordonnancement; et même pour ce type de problèmes, pour certaines types de crossover et de mutation des corrections doivent être apportées afin de garder le chromosome cohérent - consistent - (c-à-d qu'il possède une séquence d'un nombre réel).

Exemple du problème : TSP

Le problème : (cf. Annexe 1.).

Le codage : le chromosome informe de l'ordre des villes, selon lequel le VRP doit les visiteur.

- **Le codage par valeur**

Le codage direct des valeurs peut être utilisé dans des problèmes, où quelques valeurs complexes - difficiles -, comme les nombres réels, sont utilisées. L'utilisation d'un codage binaire pour ce type de problèmes est très difficile.

Dans le codage par valeur, chaque chromosome est une chaîne qui représente une certaine valeur. Le type de valeur peut varier, et les valeur peuvent être des nombres entiers, des nombres réels, ou des caractères pour certains objets complexes, par exemple:

Chromosome A 1.2324 5.3243 0.4556 2.3293 2.4545
Chromosome B ABDJEIFELSQHHEUPMDHUELQGT
Chromosome C (back), (back), (right), (forward), (left)

Le codage par valeur convient très bien pour certains problèmes spéciaux. D'un autre côté, pour ce codage il est souvent nécessaire de développer des nouveaux types de crossover et de mutations et qui sont spécifiques au problème.

Exemple du problème : Trouver les poids dans les réseaux de neurones.

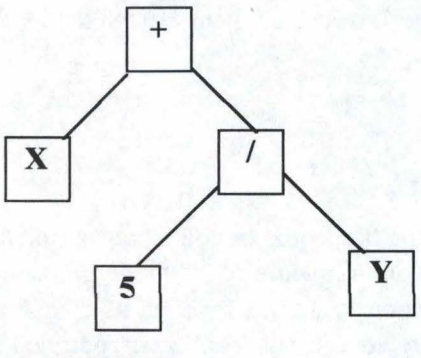
Le problème : Il y a un certain réseau neuronal avec une certaine architecture. Trouver les poids pour les entrées -inputs- des neurones afin d'apprendre le réseau - le diriger- vers un output désiré.

Le codage : Les valeurs des réels contenues dans les chromosomes représentent les poids correspondants des inputs.

• Le codage par arbre

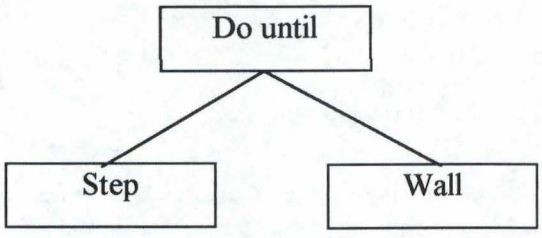
Le codage par arbre est utilisé principalement pour les programmes évolutifs ou pour des expressions dans les AG. En codage par arbre, chaque chromosome est un arbre de certains objets comme des fonctions, ou des commandes d'un langage de programmation, par exemple (ici on parcourt l'arbre en « Preorder », rgd):

chromosome A



(+X(/5Y))

chromosome B



do_until step wall

Le codage par arbre est bien pour les programmes évolutifs (PE). Le langage de programmation LISP est souvent utilisé pour cela, car les programmes en LISP sont représentés en cette forme, et ils peuvent être facilement analyser - parser - sous cette forme, et ainsi le crossover et la mutation peuvent être facilement traités.

Exemple de problème : trouver la fonction à partir d'une certaine valeur.

Le problème : certaines valeurs d'input et d'output sont données. La tâche est de trouver une fonction, qui donnera le meilleur output (ou la plus proche de ce qui est demandée) à tous les inputs.

Le codage : les chromosomes sont des fonctions représentées par un arbre.

- **Conclusion**

Un des quatre points fondamentaux qui rendent les AG différents des méthodes classiques est le fait que les AG utilisent un codage des paramètres- du problème-, et non les paramètres eux mêmes. Autrement dit, les algorithmes génétiques nécessitent le codage de l'ensemble des paramètres d'origine du problème d'optimisation en une chaîne de caractères d'alphabet et de longueur finis. Ici, nous devons distinguer les paramètres du problème (comme par exemple la variable de décision), de paramètres de l'AG comme : la probabilité ou le taux de crossover (p_c ou *crate*), la probabilité de mutation (p_m ou *mrate*), la taille de la population (popsize), ou enfin le nombre de générations (gen). Cependant, le codage de la variable de décision (et pas la variable elle-même) peut être vu comme un paramètre de l'AG, car il influence, directement ou indirectement, les performances de l'AG. En effet, les performances dépendent directement du type de codage (binaire, par valeur,...), de la longueur du chromosome, etc ; et indirectement car les opérateurs génétiques comme la fonction d'évaluation, la sélection, le crossover, la mutation, dépendent tous du codage.

Donc, pour conclure on peut dire que le codage dépend de la nature du problème et aussi de sa taille. Goldberg, dans son excellent livre ([GOL 91]) propose deux principes de base pour choisir un codage :

1) ***Le principe de la pertinence des briques élémentaires (blocs de construction)*** : l'utilisateur doit sélectionner un codage de façon à ce que les schèmes (schémas) courts et d'ordre faibles soient pertinents pour le problème sous-jacent, et relativement indépendants des schèmes aux autres positions instanciés.

2) ***Le principe des alphabets minimaux*** : l'utilisateur doit choisir le plus petit alphabet qui permette une expression naturelle du problème.

On peut démontrer que, parmi tous les codages possibles, l'alphabet binaire est celui qui offre le nombre maximal de schèmes par bit d'information (pour plus d'explication, on peut se référer au livre de Goldberg).

2.3.2. Evaluation

Nous allons étudier une fonction d'adéquation, qui évalue l'adéquation de chaque individu- ou chromosome- représenté sous forme binaire, et pour cette raison d'ailleurs on l'appelle fonction binaire d'évaluation (pour plus de détails cf. [DAV 91]). Cette fonction $\mathbf{f_6}$ est une fonction mathématique. On rencontre fréquemment ce type de fonctions dans la littérature concernant les AG. L'opération de cette fonction n'est pas simple. Décrivons un peu plus précisément cette opération.

La fonction binaire $\mathbf{f_6}$ fait trois choses. D'abord, il décode le chromosome, qui est une liste de 44 bits, en le convertissant en deux nombres réels ; puis elle fournit ces deux nombres à une fonction mathématique ; et enfin elle renvoie la valeur de cette fonction pour les deux réels. Cette valeur retournée est la valeur de l'évaluation du chromosome.

Pendant la première phase donc, $\mathbf{f_6}$ prend le chromosome et le convertit en deux nombres réels, chacun d'eux se situe entre -100 et 100. Elle effectue cette conversion en trois étapes: premièrement, elle interprète les premiers 22 bits du chromosome comme une représentation d'un entier x_1 en notation en base 2, et elle intrprète les 22 derniers bits comme un entier y_1 en notation en base 2 aussi. Deuxièmement, elle multiplie x_1 et y_1 par 0.0000476837271889898 (en notation anglaise le point remplace la virgule), ce qui fournit x_2 et y_2. Cette étape de division (multiplication par un nombre inférieur à 1) transforme (maps) l'intervalle de valeurs possibles de x_1 et y_1 (un intervalle de 0 à 2^{22}-1) en l'intervalle entre 0 et 200. Finalement, la fonction d'évaluation soustrait 100 de x_2 et y_2 pour créer x_3 et y_3, des nombres réels entre -100 et 100, comme il est illustré dans l'exemple suivant :

Chromosome:
0000101000011000000001100010101000111011 11
est divisé en:
0000101000011000000001 et 1000101010001110111011
ces chaînes de bits sont convertis de en base de 2 en base de 10 donnant ainsi x_1 et y_1 :
165377 et 2270139
ces nombres sont multipliés par :
0.00004768372718899898
pour donner ainsi x_2 et y_2 :
7.885791751335085 et 108.24868875710696
ces nombres sont diminués(par soustraction) par 100 pour donner x_3 et y_3 :
-92.11420824866492 et 8.248688757106959
l'évaluation(la fitness) du chromosome est 0.5050708, la valeur de $\mathbf{f_6}$ quand elle est appliquée à x_3 et y_3.

Dans cet exemple, le chromosome est divisé (en deux partitions) et interprété, puis divisé par une constante et shifté pour les situer entre -100 et 100. Le résultat de tout cela est que un chromosome d'une chaîne de bits est décodé produisant ainsi une paire d'entiers.

Maintenant, $\mathbf{f_6}$ calcule la valeur d'une fonction mathématique en utilisant x_3 et y_3 comme input pour cette fonction. Cette fonction qui est une version inverse de la fonction appelée $\mathbf{F_6}$ (cf.[SCH 89]), est la suivante:

$$0.5 - \{[(\sin (x^2 + y^2)^{1/2} - 0.5] / [1.0 + 0.001(x^2 + y^2)]^2\}.$$

Ici notre but est de trouver des valeurs pour x_3 et y_3 telles que si on remplace x et y dans la formule par ces valeurs, cela produit la plus grande valeur possible. Autrement dit, notre but est d'*optimiser* cette fonction.

Schaffer et al. (cf.[SCH 89]), ont choisi la $\mathbf{f_6}$ binaire pour mesurer l'efficacité des AG parce qu'elle possède plusieurs caractéristiques qui font d'elle un cas de test intéressant. D'abord, la fonction possède une seule solution optimale. Puis, les valeurs de x varient entre -100 et 100, et l'amplitude de la courbe pour n'importe quel x est la valeur retournée par $\mathbf{f_6}$. Enfin, cette fonction est symétrique par rapport à x et y (pour plus de détails, cf.[SCH 89]).

- **Exemple d'implémentation**

Pour évaluer l'adéquation- ou l'adaptation- d'une chaîne, ou d'un individu, on doit d'abord décoder la chaîne et puis utiliser une fonction qui calcule - proprement dit- la valeur de l'adaptation. Mais comme nous l'avons vu au §2.3.1."Codage"-"Conclusion", le codage dépend de la nature du problème, et par conséquent, le décodage en dépend aussi.Donc, pour tout problème, il nous faut créer une procédure qui décode une chaîne et crée un ensemble de paramètres qui soient appropriés pour ce problème. Il nous faut aussi créer une procédure qui reçoive l'ensemble de paramètres ainsi décodés et retourne la valeur

de la fonction d'adaptation associée à cet ensemble de paramètres (ici de nouveau, afin d'éviter toute confusion, on rappelle que les paramètres, ici, sont les paramètres concernant les deux procédures, et pas l'AG). Ces deux procédures, que nous appelons *decode* et *objfunc*, sont les points de contacts entre les AG et l'application.

Pour des problèmes différents, il nous faudra souvent employer des procédures de décodage différentes (bien qu'il existe des procédures standard qui se sont révélées utiles dans un grand nombre de cas), et dans différents problèmes, il nous faudra toujours utiliser des procédures différentes pour la fonction d'adaptation. Il est malgré tout utile d'examiner une procédure de décodage particulière, ainsi qu'une fonction d'adaptation particulière. Nous utilisons, ici, des codages en entier binaires positifs, et une simple fonction d'élévation à la puissance n comme fonction d'adaptation, $f(x) = x^{10}$ (ici, n = 10).

La procédure de décodage utilise la fonction *decode*. Cette fonction décode un *chromosome* unique en partant de plus bas niveau (position 1) puis en parcourant de la droite vers la gauche en additionnant 2 élevé à la puissance courante- stockée dans la variable *powerof2*- quand le bit en question est à un. La valeur cumulée, stockée dans la variable *accum*, est retournée par la fonction *decode*.Comme pour les autres implémentations, les déclarations des variables, des types, des fonctions se trouvent à la fin du §2.3.4.

```
  function decode (chrom : chromosome; lbits : integer) : real;
{la fonction decode décode une chaîne binaire en un entier positif unique ; true=1, false=0 }
  var j : integer;
      accum, powerof2 : real;
  begin
    accum := 0.0; powerof2 := 1;
    for j := 1 to lbits do begin
      if chrom[j] then accum := accum + powerof2;
      powerof2 := powerof2 * 2;
    end;
   decode := accum;
  end;
```

La fonction à optimiser utilisée ici- dans notre exemple- est une fonction puissance simple. On évalue la fonction $f(x) = (x/coeff)^{10}$. La valeur actuelle de coeff est choisie afin de normaliser le paramètre x quand la chaîne de bits de longueur lchrom = 30 a été choisie. Ainsi $coeff = 2^{30} - 1 = 1073741823$. Puisque la valeur de x a été normalisée, la valeur maximale de la fonction sera $f(x) = 1.0$ quand $x = 2^{30} - 1$, pour le cas lchrom = 30. Dans ce cas l'espace de recherche est de $2^{30} = 1{,}07 \times 10^9$ points, tandis que la valeur de la fonction varie sur l'intervalle [0,1]. Bien entendu, avec plus de 1,07 milliard de points dans l'espace, l'exploration aléatoire ou l'énumération ne devrait pas être si avantageuse, tandis que les méthodes optimisant un point à la fois ont peu de chances d'obtenir des résultats rapidement.

La programmation la plus simple de la fonction puissance- ou fonction de l'adaptation- *objfunc* est la suivante :

```
        function objfunc(x : real) : real;
{ la fonction objfunc calcule la valeur d'adaptation à partir du paramètre décodé x- f(x) = x**n }
        const coef = 1073741823.0;  { le coefficient pour normaliser le domaine }
              n = 10;               { puissance de x }
        begin objfunc := power(x/coef, n) end;
```

La fonction *power* est la suivante :

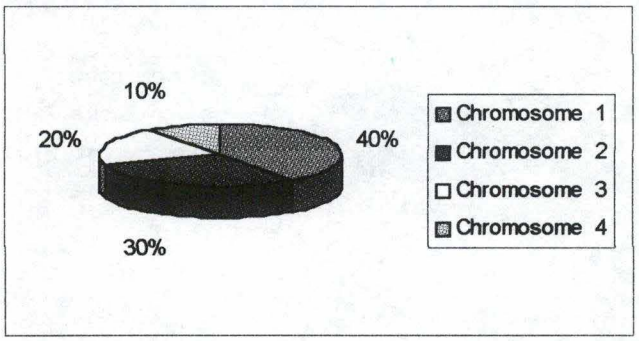
```
function power(x,y: real):real;
{ élève x à la puissance y}
begin power := exp(y*ln(x) ) end;
```

2.3.3. Sélection

Comme on l'a déjà vu en § 2.1. et § 2.2., les chromosomes sont sélectionnés pour être des parents croisés – par crossover-. Le problème est de comment sélectionner ces chromosomes. Selon la théorie de l'évolution de Darwin, les meilleurs parmi eux vont survivre et créer des descendants. Il existe plusieurs méthodes pour sélectionner les meilleurs chromosomes, par exemple, « La roue de la Fortune », la sélection de Boltzman, la sélection par tournoi - par carrousel-, la sélection par rangement ; la sélection par état stable et plusieurs autres.

- **La sélection par La roue de la Fortune**

Les parents sont sélectionnés selon leur adéquation - fitness-. Plus les chromosome sont "biens", plus leurs chances d'être choisis sont grandes. Imaginons une roue de loterie - de la Fortune - où sont placés tous les chromosomes de la population; chacun occupe une section de la roue proportionnelle à son adaptation[2], comme le voit dans la figure suivante :



Puis, une bille- ou un dé, ou par une autre méthode - est jetée pour sélectionner le chromosome .Le chromosome avec la plus grande adéquation sera sélectionné plus de fois que les autres. Cela peut être simulé par l'algorithme suivant :

1. [**somme**] Calculer la somme des adéquations de tous les chromosomes de la population - somme **s**
2. [**select**] Générer un nombre aléatoirement qui appartient à l'intervalle **(0,s)** - **r**.
3. [**loop**] Aller à travers la population , et somme (additionne) les adéquations à partir de 0 - somme **s**

Quand cette somme **s** est plus grande que **r**, Stop et Retourner le chromosome où on se trouve.

Remarque: l'étape 1 est , évidement, exécutée une seule fois pour chaque population.

- **La sélection par rangement**

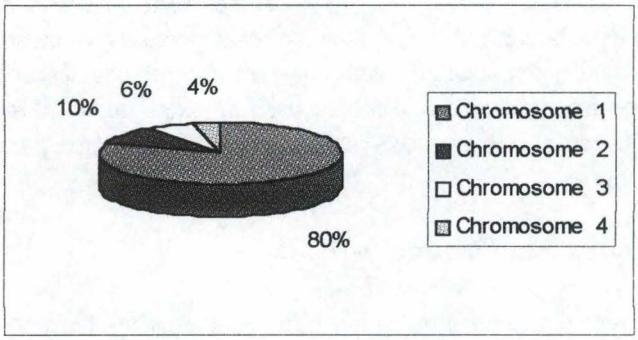
La sélection précédente va avoir des problèmes si les adéquations sont très différentes- très espacées- les unes des autres. Par exemple, si la meilleure adéquation d'un chromosome représente 90 % de la totalité de la roue, alors les autres auront très peu de chances d'être sélectionnés.

---

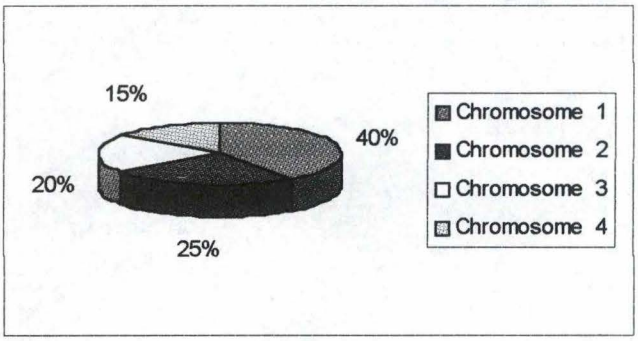
[2] adaptation, adéquation, aptitude tous sont des synonymes pour traduire le mot"fitness" en anglais, comme crossover, crossing-over, croisement, accouplement sont eux aussi des synonymes.

La sélection par rangement range d'abord la population, et puis chaque chromosome reçoit une adéquation de ce rangement- ou du nouveau ordonnancement-. Le pire- le plus mauvais- reçoit 1 comme adéquation, le deuxième moins pire reçoit 2, etc... et le meilleur reçoit N (N étant le nombre de chromosomes de la population).

On peut remarquer dans les deux figures suivantes, comment la situation est-il avant et après le changement des adéquations en numéros d'ordre.



La situation avant le rangement (le graphique de l' adéquation).



La situation après le rangement (le graphique des numéros d'ordre).

Après cela tous les chromosomes ont des chances d'être sélectionnés. Mais cette méthode peut conduire au ralentissement de la convergence, car les meilleurs chromosomes ne diffèrent pas beaucoup les uns des autres.

Remarque :

Nous pouvons à partir de la sélection par rangement aller -par un processus de généralisation- vers les mécanismes généraux dont, cette sélection est un cas particulier. Ces mécanismes sont les mécanismes de changement d'échelle.

Le changement d'échelle des fonctions à optimiser est une pratique commune qu'on fait afin de garder un niveau de compétitivité adéquat tout au long de la simulation. Sans changement d'échelle, il y a au début une tendance à la domination du processus de sélection par les super-individus. Dans ce cas,les valeurs de la fonction doivent être réduites pour empêcher que ces super-chaînes (ou super-chromosomes) "envahissent" la population. Plus tard, quand la population a atteint un certain degré de convergence, la compétition entre les différents individus devient moins forte, et la simulation tend à perdre son efficacité. Dans ce cas les valeurs de la fonction doivent être amplifiées pour accentuer les différences entre les individus de la population pour continuer à récompenser les meilleurs- ce qu'on appelle en anglais "windowing", ou fenêtrage-. Il existe plusieurs méthodes de changement d'échelle parmi lesquelles, on trouve :

1. **Le changement d'échelle linéaire**
2. **La troncature en sigma(s)**
3. **Le changement d'échelle en puissance**.

Dans le *changement d'échelle linéaire*, on calcule simplement l'adéquation transformée f' à partir de l'adéquation brute (la valeur de la fonction à optimiser) en utilisant une équation linéaire de la forme :

$$f' = af + b$$

Dans cette équation, les coefficients a et b sont choisis d'habitude pour faire deux choses : maintenir l'égalité des moyennes d'adéquation brute et transformée, et conduire l'adéquation transformée maximale à être un multiple déterminé (généralement le double) de l'adéquation moyenne. Ces deux conditions permettent de s'assurer que les individus de valeur moyenne sont copiés une fois en moyenne, et les meilleurs sont copiés un nombre de fois égal au multiple choisi. Pour le changement d'échelle linéaire, il faut faire attention d'éviter des valeurs d'adéquation transformées négatives.

Le changement d'échelle linéaire obtient de bons résultats sauf quand son emploi est limité par le risque de valeurs d'adéquations négatives. Ce problème apparaît le plus souvent vers la fin d'une exécution quand la plupart des individus sont bien adatés, mais qu'il reste quelques "monstres" de valeurs très faible. Pour éviter ce problème, on a suggéré l'utilisation de la variance de la population pour prétraiter l'adéquation brute avant le changement d'échelle. Dans cette procédure, que l'on appelle la *troncature en sigma*(s) à cause de l'utilisation de l'écart-type de la population, on soustrait une constante de l'adéquation brute comme suit :

$$f' = f - (\bar{f} - c*\sigma)$$

Dans cette équation, la constante c est un coefficient de l'écart-type choisi arbitrairement (des valeurs raisonnables sont comprises entre 1 et 3). Les résultats négatifs ($f' < 0$) sont transformés en zéros. Après la trancature en sigma, la transformation d'adéquation peut être poursuivie sans les dangers de résultats négatifs.

On a suggéré un *changement d'échelle en puissance* pour laquelle l'adéquation transformée prend la valeur d'une certaine puissance de l'adéquation brute f :

$$f' = f^k .$$

- **Exemple d'implémentation**

Le "windowing" est peut être la forme la plus simple du changement d'échelle. Pour le réaliser, on commence par calculer comme d'habitude les valeurs d'adéquation (de fitness), en gardant trace (en sauvant) de la plus petite valeur de fitness. Puis, on soustrait cette valeur minimum (minf) de toutes les valeurs de fitness (y compris de la valeur minimum elle-même), ainsi on ajuste le vecteur de fitness par rapport à la base zéro (en fait, on a effectué ici un changement d'échelle linéaire, où a = 1,et b = –minf). On peut aussi soustraire une valeur légèrement plus petite que la valeur minimum pour s'assurer que tous les chromosomes vont avoir une chance de se reproduire. Dans l'implémentation suivante, on teste si l'option "changement d'échelle" a été activé ; si oui, alors on soustrait (minf), on ajoute 1, puis on met le résultat au carré, et cela, bien entendu, pour chaque valeur du vecteur de fitness. On ajoute 1 pour être sûr que chaque chromosome a une chance de se reproduire, tandis que le fait de mettre au carré les valeurs déjà "fenêtrées" (windowed fitness values) augmente les chances reproductives des meilleurs chromosomes. Ici on peut dire, qu'on a d'abord effectué un changement d'échelle linéaire, puis un changement d'échelle en puissance (k=2). Nous avons choisi une implémentation en C++ (ou en C) pour qu'il ne soit pas toujours en PASCAL.

```
if (scale)                          // si on a activé le changement d'échelle
        {
          ++minf;                   // s'assurer que le minimum de fitness est 1
      recalculer le total( la somme totale) de fitness pour reflléter le changement d'échelle
                totf = 0L;
                for ( i = 0; i < POP_SZ; ++i)       // POP_SZ est la taille de la population
                    {
                       fit[i] -= minf;      // soustraire le minimum de chaque valeur de fitness
                       fit[i] *= fit[i];    // mettre le résultat au carré
                       totf  += fit[i];     // ajouter au total de fitness
                    }
        }.
```

- **La sélection par l'état-stable (steady-state selection)**

Ce n'est pas une méthode particulière ou spéciale pour sélectionner les parents. L'idée principale de cette sélection est que la plus grande partie- la majorité- des chromosomes doivent survivre jusqu'à la génération suivante.

Les AG procèdent alors de la façon suivante. A chaque génération quelques (bons- avec bonne adéquation) chromosomes sont sélectionnés pour créer la nouvelle descendance. Puis quelques (mauvais- avec mauvaise adéquation) chromosomes sont éliminés et la nouvelle descendance est placée à leur place. Le reste de la population survit jusqu'à la nouvelle génération.

- **L'élitisme ou la sélection élitiste**

L'idée de l'élitisme a été déjà décrit. Quand une nouvelle génération est créée par crossover et mutation, il y a des fortes chances qu'on va perdre le meilleur chromosome. La stratégie élitiste consiste à copier d'abord le meilleur chromosome (ou plusieurs chromosomes) dans la nouvelle population (la population suivante). Le reste se déroule selon la façon classique. L'élitisme peut augmenter très rapidement la performance de l'AG, car il prévient de perdre la meilleure solution qu'on a trouvé.

- **La sélection par le tournoi stochastique**

Dans cette méthode (que certains auteurs qualifient de méthode d'évaluation, ou "ranking method"), les probabilités de sélection sont calculées comme d'habitude et des paires successives d'individus- ou plusieurs individus, cela dépend du paramètre du tournoi (cf.§2.3.5 "D'autres paramètres"), sont tirés au hasard grâce à la sélection par roue de loterie. Après avoir tirer une paire de- ou plusieurs-chaînes ou chromosomes, la chaînes ayant l'adéquation la plus élevée est déclarée vainqueur, elle est ajoutée à la nouvelle population, et une nouvelle paire de- ou plusieurs- chaînes sont tirées. Ce processus continue jusqu'à ce que la population soit remplie,ou on a atteint un certain nombre- fixé à l'avance- d'individus qui peuvent se reproduire.

- **Exemple d'implémentation**

On peut programmer la sélection par une fonction *select* qui effectue une recherche linéaire sur une roue de loterie dont les sections ont une taille proportionnelle aux valeurs d'adéquation des chaînes (comme on l'a déjà expliqué ci-dessus). Dans le code présenté ci-après, on voit que *select* retourne l'index qui correspond à l'individu sélectionné dans la population. A cette fin, la somme partielle des valeurs d'adéquation est cumulée dans la variable réelle *partsum*. La variable réelle *rand* contient la position où la roue s'est arrêté après avoir été lancée, selon le calcul :

$$rand := random \times sumfitness \qquad (1)$$

On suppose l'existence d'une procédure *random* qui retourne un nombre réel pseudo-aléatoire compris entre zéro et un (une variable aléatoire uniforme sur l'intervalle réel [0,1]). On peut avoir cette variable en PASCAL, par exemple, par RANDOM/MAXINT, ou par la formule

alea := MIN + RANDOM MOD(MAX-MIN+1)

ou par un autre moyen; tandis que en C on peut l'avoir, par exemple, par :

rand( ) / RAND_MAX.

sumfitness est la somme des adéquations de la population qu'on peut calculer facilement par une boucle de type :

```
sumfitness := pop[1].fitness;
for j := 2 to popsize do
sumfitness := sumfitness + pop[j].fitness;
```

Donc pour calculer la variable rand en (1), la somme des adéquations est multipliée par le nombre pseudo-aléatoire normalisé généré par *random*. Finalement, la boucle *repeat-until* explore la roue de loterie jusqu'à ce que la somme partielle soit supérieure ou égale à la valeur d'arrêt rand. La fonction retourne alors dans *select* la valeur de l'index de population *j*.

```
function select (popsize : integer; sumfitness : real; var pop : population) : integer;
{ sélectionner un individu par la sélection par la roue de la fortune(loterie)}
var     rand, partsum : real;      { un point aléatoire sur la roue, la somme partielle}
    j : integer;                   { l'index de population}
begin
 partsum := 0.0; j := 0;           { on met le compteur et l'accumulateur à zéro}
 rand := random * sumfitness; {on calcule le point sur la roue en utilisant le nombre aléatoire ∈ [0,1]}
 repeat                            { trouver la section }
  j := j + 1;
  partsum := partsum + pop[j].fitness;
 until (partsum >= rand) or (j = popsize);
 select := j;                      { retouner le numéro de l'individu }
end;
```

Ceci est peut-être la manière la plus simple de programmer la sélection. On peut aussi coder l'opérateur de façon plus performante (une exploration binaire- ou recherche dichotomique -devrait accélérer le processus), et il y a un grand nombre de possibilités différentes pour choisir les descendants avec un biais approprié en faveur des meilleurs (cf. §2.3.3. "conclusion").

- **Conclusion**

Plusieurs chercheurs ont étudié d'autres possibilités de sélection, en essayant de réduire les erreurs stochastiques liées à la sélection par roue de loterie.Citons en quelques méthodes (pour plus de détails cf.[GOL 91]) :

1. L'échantillonage déterministe
2. Echantillonage stochastique de la partie restante sans remplacement
3. Echantillonage stochastique sans remplacement
4. Echantillonage stochastique de la partie restante avec remplacement
5. Echantillonage stochastique avec remplacement
6. Tournoi stochastique (évaluation de Wetzel)

En fait l'Echantillonage stochastique avec remplacement est un autre nom de la sélection par roue de loterie, tandis que le tournoi stochastique, nous l'avons déjà expliqué au §2.3.3. Des travaux et des

études avec des fonctions de test habituelles ont été mené sur ces différentes méthodes, et ont conclu aux deux résultats suivants :

a) L'infériorité fondamentale de la simple sélection par roue de loterie par rapport aux autres méthodes, ce qui confirme des résultats qu'on avait déjà observé dès 1975, tandis que les différences de performances entre les cinq autres mécanismes sont faibles, et on a pu- en 1985- recommander une plutôt qu'une autre.
b) Mais une étude ultérieure concernant les méthodes 2 et 3 ("Echantillonage stochastique de la partie restante sans remplacement", "Echantillonage stochastique sans remplacement"), a montré la supériorité de la méthode 2 par rapport à la méthode 3. Il s'en suivit que la procédure de sélection stochastique de la partie restante sans remplacement s'est trouvée largement utilisée dans les applications suivantes.

2.3.4. Reproduction avec croisement et mutation

Le crossover et la mutation sont deux opérateurs essentiels des AG. Les performances des AG dépendent fortement d'eux. Les types et implémentations de ces opérateurs dépendent du codage et du problème. Il y a plusieurs façons de faire le crossover et la mutation. Dans ce paragraphe on va donner quelques exemples et suggestions de ces façons pour plusieurs codages.

- **Codage binaire**

Le crossover par un point (single point crossover) :

Un point de crossover est sélectionné ; la chaîne binaire du début de chromosome jusqu'au point de crossover est copiée d'un parent, et le reste est copié du deuxième parent.

1101011 + 11011**111** = **1101111**

Le crossover par deux points (two points crossover) :

Deux points de crossover sont choisis ; la chaîne binaire du début du chromosome jusqu'au premier point est copié du premier parent, la partie du premier point jusqu'au deuxième est copiée du deuxième parent, et le reste est copié du premier parent :

1100101**1** + 11**011111** = **11011111**

et le deuxième enfant recevra le reste :

11001011

Le crossover uniforme :

Tandis que le crossover par deux points peut combiner deux schémas (en 3 parties différentes) qui se trouvent chez les deux parents et que le crossover par un point ne peut pas le faire,comme par exemple:

chromosome$_1$ **11**011001011**11**
chromosome$_2$ 000**10**110**111**00

où on peut constater que le crossover de 1-point ne peut pas combiner les deux schémas, tandis que le crossover de 2-points le peut de la façon suivante:

parent$_1$ **1101**|100101|**1011**
parent$_2$ 0001|**011011**|1100

enfant$_1$ **11010110111011**
enfant$_2$ 00011001011100

Il y a des situations où même le crossover par deux points ne peut pas, lui aussi, le faire comme par exemple,Si les deux parents sont:

parent$_1$ ABCDEFG
parent$_2$ HIJKLMN

Et le schéma recherché est: AIJDEMN. Alors aucun crossover par deux points ne peut combiner ces schémas pour avoir le schéma recherché. Pour cette raison Syswerda (cf.[SYS 89]) a proposé le crossover uniforme. Par exemple:

parent$_1$ 1001011
parent$_2$ **0101101**

et le modèle recerché est 1101001

Le crossover uniforme donne:

enfant$_1$ 1**001101**
enfant$_2$ **0**101011

Il fonctionne de la façon suivante: deux parents sont sélectionnés et deux enfants sont produits. Pour chaque position de bit de deux enfants, on décide aléatoirement lequel de deux parents va contribuer par son bit- plus exactement sa valeur- à cet enfant. Dans notre exemple précédent, pour chaque bit de position- d'une certaine position- le modèle indique lequel de deux parents qui va contribuer par sa valeur- la valeur de son bit de cette position- dans cette position de l'enfant$_1$. Le deuxième enfant reçoit la valeur de bit de cette position de l'autre parent- elle qui n'était pas choisie-.

Le crossover uniforme est différent du crossover de 1-point et du crossover de 2-points en plusieurs points. D'abord, la place de l'encodage d'une certaine caractéristique- d'un certain trait- dans un chromosome n'est pas important. Avec le crossover de 1-point ou 2-points, plus qu'il y a des bits qui interviennent dans un schéma, moins vraisemblablement que ce schéma sera passé intact à un enfant. Cependant, un danger potentiel caractéristique du crossover uniforme est le fait qu'il provoque des grands dégâts à tout se qui est bon dans un chromosome.

Deuxièmement, l'action des opérateurs de crossover de 1-point et de 2-points est plus locale que celle de l'uniforme, et ils préservent, vraisemblablement, plus les bonnes caractéristiques d'autant que celles-ci sont encodées d'une façon compacte. Syswerda a fait des expériences qui ont montré que pour certains problèmes, la capacité du crossover uniforme à combiner les bonnes caractéristiques, peu importe leurs positions, l'emporte sur la dévastation globale que ce crossover peut, probablement, causer quand on l'utilise sur des chromosomes radicalement dissemblables.

Le crossover arithmétique :

Une certaine opération arithmétique est exécutée pour créer un nouveau descendant.

11001011 + 11011111 = 11001001 (AND)

La mutation :

C'est l'inversion d'un (ou plusieurs) bits. Les bits sélectionnés sont inversés.

11001001 => 10001001

- **Exemple d'implémentation**

La procédure *crossover* que nous avons implémenté prend pour paramètres- paramètres de la procédure et pas de l'AG- deux parents *parent1* et *parent2*, et génère deux chaînes appelées *child1* et *child2*. Les probabilités de crossover et de mutation, *pcross* et *pmutation*, sont passées à *crossover*, ainsi que *lchrom* la longeur de chaîne, *ncross* un compteur pour les mutations.

Remarque1 : les déclarations des types, des variables globales et des fonctions aléatoires se trouvent au début de l'exemple §2.3.5."Exemple d'implémentation de la procédure *génération*".
Remarque2 : Dans cette procédure, on effectue le crossover et la mutation ensemble, on aurait pu les séparer comme on va le voir dans l'implémentation en C++. Ici c'est un crossover simple en 1 point.

```
procedure crossover (var parent1, parent2, child1, child2 : chromosome;
                     var lchrom, ncross, nmutation, jcross : integer;
                     var pcross, pmutation : real);
{ croiser deux chaînes parents, placer le résultat dans 2 chaînes enfants}
var j : integer;
begin
 if (pcross) then begin   { faire le crossover avec probabilité pcross}
   jcross := rnd(1,lchrom-1);   { déterminer le point de crossover aléa. entre 1 et lchrom-1}
   ncross := ncross + 1;       { incrémenter le compteur de crossover}
 end else                      { sinon, pas de crossover, préparer la chaîne pour la mutation}
   jcross := lchrom;
          { premier échange, parent1 et child1,parent2 et child2}
  for j := 1 to jcross do begin
   child1[j] := mutation(parent1[j], pmutation, nmutation);
   child2[j] := mutation(parent2[j], pmutation, nmutation);
  end;
          { deuxième échange, parent1 et child2,parent2 et child1}
 if jcross <> lchrom then   { ne pas faire si le site du crossover est lchrom = pas de crossover}
  for j := jcross+1 to lchrom do begin
      child1[j] := mutation(parent2[j], pmutation, nmutation);
      child2[j] := mutation(parent1[j], pmutation, nmutation);
  end;
end;
```

Quelques explications. Au début de la procédure, on détermine si le crossover va être effectué sur la paire de chromosomes parents considérés. Plus précisément, on tire à pile ou face avec une pièce biaisée- sinon on aura toujours ±1/2 pour pile, et ±1/2 pour face- qui propose face avec une probabilité *pcross*. Le tirage est simulé dans la fonction booléenne *flip*, qui elle-même appelle le générateur de nombres pseudo-aléatoires *random*. Si un crossover est décidé, un lieu de crossover est sélectionné. Ceci est fait dans la fonction *rnd*, qui retourne un entier pseudo-aléatoire compris entre les limites inférieure et supérieure(entre 1 et *lchrom*-1). Si aucun crossover n'a lieu, le lieu de croisement(ou de crossover) se voit affecté la valeur *lchrom*(la longeur de chaîne pleine *l*), si bien qu'une mutation bit par bit aura lieu malgré l'absence de crossover. Enfin, l'échange partiel du crossover a lieu dans les boucles for-do à la fin du code. La première boucle s'occupe du transfert partiel de bits entre parent1 et child1 et entre parent2 et child2. La seconde se charge du transfert et de l'échange partiel de matériel génétique entre parent1 et child2, et entre parent2 et child1. Dans tous les cas, une mutation bit à bit est effectuée par la fonction booléenne (ou "allèléenne") *mutation.*

La mutation en un point est effectuée par *mutation*. Cette fonction utilise *flip* (la pièce biaisée) pour décider s'il faut changer une valeur vraie à une valeur fausse (de 1 à 0) ou vice versa. Bien entendu, la fonction *flip* ne retournera face (vrai) que seulement *pmutation* pour-cent du temps du fait de l'appel

par *flip* du générateur de nombres aléatoires *random*. La fonction gère le nombre de mutations en incrémentant la variable *nmutation*. Comme pour la reproduction, il y a moyen d'amélorer cet opérateur de mutation simple. Par exemple, il serait possible d'éviter beaucoup de générations de nombres aléatoires en décidant quand aura lieu la mutation suivante plutôt que d'appeler *flip* à chaque fois.

```
function mutation (alleleval : allele; pmutation : real; var nmutation : integer) : allele;
{ faire la mutation d'un allèle avec une probabilité pmutation, compter le nombre de mutations}
var  mutate : boolean;
begin
  mutate := flip(pmutation);        { jeter la pièce biaisée}
  if mutate then begin
    nmutation := nmutation + 1;
    mutation := not alleleval;     { changer la valeur du bit}
  end else
    mutation := alleleval;          { sans changement}
end;
```

En C++(ou en C) la procédure crossover sera implémentée de la façon suivante :

```
        // la reproduction par crossover
  if (cross && ( (float(rand( )) / float(RAND_MAX) ) < crate)   // si le nombre généré aléatoirement
                                                                // est inférieur au taux de crossover
   {
        mask = 0xFFFFFFFFL << (int) ((float(rand( ) / float(RAND_MAX)) * 32.0F);
        newpop[i] = ( pop[p1] & mask) | (pop[p2] (~mask));
        ++i;
        newpop[i] = (pop[p1] & (~mask)) | (pop[p2] & mask);
   }
  else{         // pas de reproduction par crossover, simplement copiage
       newpop[i] = pop[p1]; ++i; newpop[i] = pop[p2];
     }
```

Quelques mots d'explication. D'abord, on teste **si** l'option "crossover" est activée ET (et logique) si le nombre aléatoire, qui représente la probabilité de faire un crossover (entre 0 et 1) est inférieure au taux de crossover fixé à l'avance *crate*- c'est un paramètre d'input-, **alors** on crée un masque de bits à 1, de longueur égal à la longueur du chromosome- ici 32 bits-, puis on effectue un shift de bits à gauche d'un nombre entier aléatoire (entre 0 et 32), et les bits qui sont sortis par la gauche seront remplacés à droite du masque par des bits à 0; ainsi notre masque ressemblera après le shift à :
11.....1100.....0, tandis que le ~masque (ou l'inverse du masque) ressemblera à :
00.....0011.....1, et puis en faisant le AND de bit (bitwise AND), entre le premier parent et le masque, on obtient la première partie du premier parent (car 1 AND bit = bit, peu importe la valeur du bit), tandis que sa deuxième partie est à 0,et de la même façon on applique le AND entre le parent2 et l'inverse du masque, pour obtenir la deuxième partie du parent2 (par le même raisonnement sur le AND),tandis que sa première partie est à 0; et enfin, on applique le OR de bit (bit OR 0 = bit) pour obtenir le premier enfant dont la première partie vient du parent1, et la deuxième partie vient du parent2. Et de la même façon, on obtient avec les parties restantes l'enfant2. Exemple :
A|B≡parent1; C|D≡parent2; 1|0≡masque; 0|1≡~masque
A|B AND 1|0 ⇒ A|0 ; C|D AND 0|1 ⇒ 0|D
A|0 OR 0|D ⇒ A|D ≡enfant1
A|B AND 0|1 ⇒ 0|B ; C|D AND 1|0 ⇒ C|0
0|B OR C|0 ⇒ C|B ≡enfant2

Sinon (si "crossover" est désactivée **OU** la probabilité de faire le crossover est ≥ *crate*) pas de crossover, et la reproduction consiste simplement à copier les parents dans les enfants.

La mutation en C/C++ est le code suivant :

```
    // mutation
If (mutate
  && ((float(rand( ) / float(RAND_MAX) < mrate))
  {
   // sélectionner le bit à changer
   mask = 1L << (int)((float(rand( ) / float(RAND_MAX)) * 32.0F);
  // changer le bit
     if (newpop[i] & mask)
        newpop[i] &= ~mask;
     else
        newpop[i] |= mask;
     }
}
```

Quelques mots d'explication. D'abord, on teste si l'option"mutation" est activée ET si la probabilité de faire une mutation- représentée par un nombre aléatoire- est inférieure au taux- fixé au début du programme par un input- de mutation *mrate*. Si c'est le cas, alors on crée un masque de 32 bits, un bit à 1 et les 31 à 0(00.....01) qui est égal à 1L; puis on déplace ce bit à 1, par un shift à gauche- comme nous l'avons fait à la procédure "crossover"- d'un certain nombre entier aléatoire de bits- plus exactement de positions de bits-, et ainsi, par exemple, le masque, après un shift, devient : (00....010...0). Puis, on applique le AND entre la nouvelle chaîne- ou l'enfant, car on exécute la mutation juste après le "crossover"- et le masque, et on teste le résultat de l'AND : si c'est vrai- ou plus exactement 1- alors on exécute l'instruction : newpop[i] &= ~mask; ou autrement dit,
newpop[i] = newpop[i] & (~mask). Digitalement, on peut l'écrire de la façon suivante :
Le masque : (00....010...0); la nouvelle chaîne : (**....*0*.....*), où "*" veut dire peu importe la valeur du bit, 0 ou 1. Pour que le résultat soit vrai, il faut que la nouvelle chaîne soit : (**....*1*...*), car : (**....*1*...*) AND (00....010...0) ⇒ (00....010...0), donc le résultat est vrai ; donc on exécute l'instruction newpop[i] &= ~mask; le non masque(~mask) étant: (11....101...1) :
(**....*1*...*) & (11....101...1) ⇒ (**....*0*...*) ; et enfin newpop[i]≡(**....*0*...*), et ainsi on voit que le bit qui était à 1 en newpop[i] est devenu, après le if-vrai et l'instruction, à 0.

Maintenant, si le résultat du AND est faux, alors newpop[i] était : (**....*0*...*) et pas (**....*1*...*) (comme dans le cas du résultat de AND vrai). Dans ce cas, on exécute l'instruction :
newpop[i] = newpop[i] OR mask :
(**....*0*...*) OR (00....010...0) ⇒ (**....*1*...*); et newpop[i]≡(**....*1*...*). Ainsi, on remarque que le bit qui était à 0, est devenu à 1. Donc, dans les deux cas, on a changé la valeur du bit sélectionné.

- **Codage à permutation**

crossover :

Le crossover par un point :

Un point de crossover est sélectionné ; jusqu'à ce point la permutation est copiée du premier parent, puis le deuxième parent est examiné et si un nombre n'est pas encore dans le descendant alors il y est ajouté.
Remarque : il y a plusieurs façons de produire le reste après le point du crossover.

(**123456**789) + (**453689**721) = (**123456**897)

La mutation :

Le changement d'ordre. Deux nombres sont sélectionnés et échangés

(123456897) => (183456297)

- **Codage par valeur**

Le crossover :

Tous les crossovers du codage binaire peuvent être utilisés ici.

La mutation :

On ajoute un petit nombre (à la valeur réelle encodée)- on choisit des valeurs à ajouter ou à soustraire.

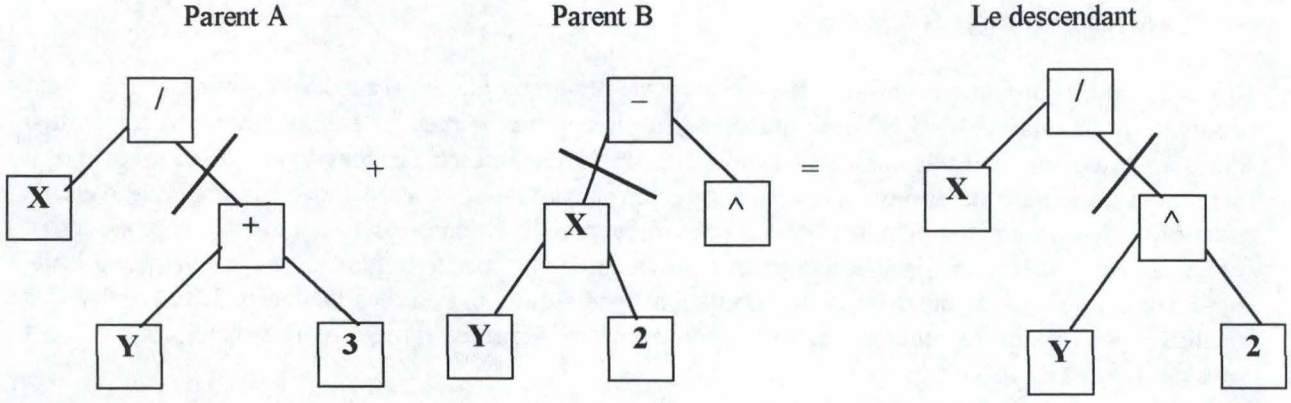
(1.29 5.68 **2.86** **4.11** 5.55) => (1.29 5.68 **2.73** **4.22** 5.55)

- **Codage par arbre**

Le crossover :

Le crossover par arbre :

Dans les deux parents, un point de crossover est sélectionné; les parents sont divisés en ces points, et ils échangent le parties en-dessous de ces points, afin de produire un descendant.



La mutation :

Ici c'est un changement d'opérateur ou du nombre ; les nœuds sélectionnés sont échangés.

2.3.5. Paramètres des AG

Il y a trois paramètres essentiels des AG : la probabilité de crossover, la probabilité de mutation- on appelle aussi ces deux probabilités des taux-, et la taille de la population en une génération.

- **La probabilité de crossover**

Elle nous informe de la fréquence de l'exécution du crossover. Quand il n'y a pas de crossover, alors le descendant est la copie exact des parents. Si il y a crossover, le descendant est constitué des parties de ses parents. Si le taux de crossover est 100 %, alors tous les descendants sont faits par crossover. S'il est de 0 %, alors la nouvelle population est constituée de copies exactes des chromosomes de l'ancienne génération (mais cela ne veut pas dire que la nouvelle génération est la même que l'ancienne !).

Le crossover est fait dans l'espoir que les nouveaux chromosomes vont avoir des bonnes parties des anciens, et peut-être les nouveaux seront meilleurs que les anciens. Toutefois, il est bon de laisser une certaine partie de la population survivre jusqu'à la population suivante.

- **La probabilité de mutation**

Elle nous informe de la fréquence avec laquelle des parties d'un chromosome seront mutées. Si il n'y a pas de mutation, le descendant est pris après le crossover- ou il est copié- sans aucun changement. Si la mutation est exécutée, une partie du chromosome est changée. Si la probabilité est de 100 %, tout le chromosome est changé, et si de 0 %, rien n'est changé.

La mutation est faite pour empêcher l'AG de tomber sur un extremum local, mais on ne doit pas souvent utiliser la mutation, car cela peut changer l'AG en recherche aléatoire.

- **Le nombre d'individus dans une population (en une génération)**

Si il y a très peu de chromosomes, l'AG n'a que peu de possibilités pour exécuter le crossover, et seulement une petite partie de l'espace de recherche est explorée. D'un autre côté, s'il y a beaucoup trop de chromosomes, cela va ralentir l'AG. Les recherches montrent que au-delà d'une certaine limite (qui dépend essentiellement du problème, surtout de la longueur l du chromosome) il devient inutile d'augmenter la taille de la population, car cela ne résout pas le problème plus vite.

- **D'autres paramètres**

Il y a des autres paramètres qui jouent un rôle moins important que les trois cités ci-dessus, comme par exemple, **le paramètre de l'élitisme** qui détermine le nombre de meilleurs individus d'une population qui seront replacés dans la population suivante; ou **le paramètre de survivants de crossover** qui détermine le nombre de survivants- à la population suivante-parmi les 4 individus impliqué dans un crossover. Ce paramètre peut prendre la valeur "enfants", et dans ce cas les deux enfants seront replacés- sans altération- dans la population suivante. Cette option conduit à une convergence lente, mais à une plus grande diversité de la population; tandis que l'option "les meilleurs deux" replace les meilleurs ceux parmi les quatre, ce qui conduit à une convergence rapide, et les bonnes solutions ont plus de chances de survivre.

Enfin, le dernier paramètre- ce n'est pas une liste exhaustive de tous les paramètres- qu'on va citer ici- est **le paramètre du tournoi** - le tournoi est une méthode de sélection qu'on a déjà abordé en 2.3.3- . Un tournoi consiste à sélectionner aléatoirement un certain nombre - déterminé par ce paramètre - d'individus de la population (un sous-ensemble de la population), et puis on compare leurs adéquations. Le meilleur parmi eux peut se reproduire, et ainsi on recommence un autre tournoi jusqu'à on a atteint, par exemple, un certain nombre- fixé à l'avance- d'individus qui peuvent se reproduire.

- **Exemple d'implémentation**

Nous allons présenter ici les déclarations des types de données, de variables globales, les définitions des trois fonctions de choix aléatoire, la procédure *génération* où on crée un nouvelle population à partir de la population précédente- en utilisant *select, crossover, mutation, decode, objfunc* qu'on les a déjà tous vu-, la procédure *stastistics* qui calcule quelques statistiques sur la population, et enfin le programme principal. En fait, il nous reste que les définitions de ces trois fonctions citées ci-dessus, et les procédures d'entrées-sorties et quelques autres petites procédures utilitaires, mais le manque d'espace nous oblige à se limiter à l'essentiel, tandis que le reste ce n'est pas très difficile de l'implémenter.

```
                        { Les déclarations des types de données }
const  maxpop       = 100;
       maxstring    = 30;

type   allele       = boolean;   { un bit d'une certaine position }
       chromosome   = array[1..maxstring] of allele;  { chaîne de bits }
       individual   = record
                        chrom : chromosome;  { génotype  = structure; une chaîne de bits }
                        x : real;            { phénotype = entier non-signé }
                        fitness : real;      { valeur de la fonction de l'objectif }
                        parent1, parent2, xiste : integer;  { les parents, et point de crossover }
                      end;
       population   = array[1..maxpop] of individual;

                        {Les déclarations de variables globales }
var    oldpop, newpop : population;              { deux populations sans chevauchement }
       popsize, lchrom, gen, maxgen : integer;   { variables entières globales }
       pcross, pmutation, sumfitness : real;     { variables réelles globales }
       nmutation, ncross : integer;              { variables entières pour les statistiques }
       avg, max, min : real;                     { variables réelles pour les statistiques }
```

Les défintions des fonctions de choix aléatoire :

random retourne un nombre réel pseudo-aléatoire compris entre zéro et un(une variable aléatoire uniforme sur l'intervalle réel [0,1]).

flip retourne un booléen de valeur vraie avec une probabilité donnée(une variable aléatoire de Bernouilli).

rnd retourne une valeur entière comprise entre des limites inférieures et supérieures(une variable aléatoire uniforme sur un ensemble d'entiers adjacents).

```
{ La procédure génération crée une nouvelle population à partir de la population précédente, en}
{utilisant select, crossover et mutation. Note : génération suppose que popsize est un nombre pair}

procedure generation;
var  j, mate1, mate2, jcross : integer;
begin
 j := 1;
 repeat        { select, crossover et mutation jusqu'à que newpop soit remplie}
  mate1 := select(popsize, sumfitness, oldpop);  { choisir une paire d'individus ou un couple }
  mate2 := select(popsize, sumfitness, oldpop);
  { crossover et mutation - la mutation est imbriquée dans le crossover }
  crossover(oldpop[mate1].chrom, oldpop[mate2].chrom,
            newpop[j].chrom, newpop[j + 1].chrom,
            lchrom, ncross, nmutation, jcross, pcross, pmutation);
```

```
{ décoder la chaîne, évaluer la fitness, et enregistrer la filiation des deux enfants}
  with newpop[j] do begin
     x := decode(chrom, lchrom);
     fitness := objfunc(x);
     parent1 := mate1;
     parent2 := mate2;
     xiste   := jcross;
  end;
  with newpop[j + 1] do begin
     x := decode(chrom, lchrom);
     fitness := objfunc(x);
     parent1 := mate1;
     parent2 := mate2;
     xiste   := jcross;
  end;
  { incrémenter l'index de la population }
  j := j + 2;
 until j > popsize
end;
```

Dans la procédure *génération*, on commence par un indice d'individu j = 1 et on continue jusqu'à ce que la taille de la population *popsize* ait été dépassée, on choisit une paire d'individus, *mate1* et *mate2*, en appelant *select*. On fait se croiser et muter les chromosomes en utilisant *crossover*(qui contient les appels nécessaires à *mutation*). Pour finir en beauté, on décode la paire de chromosomes, on calcule les valeurs de la fonction à optimiser(la fonction d'adaptation), et on incrémente l'indice de population *j* de 2.

```
{ La procédure statistics calcule quelques statistiques sur la population }

procedure statistics( popsize : integer;
                      var max, avg, min, sumfitness : real;
                      var pop : population );
   { calculer des statistiques sur la population }
var j : integer;
begin
  { initialiser }
  sumfitness := pop[1].fitness;
  min        := pop[1].fitness;
  max        := pop[1].fitness;
     { boucler pour max, min, sumfitness }
  for j := 2 to popsize do with pop[j] do begin
     sumfitness := sumfitness + fitness;    { faire la somme de fitness }
     if fitness > max then  max := fitness;   { nouveau max }
     if fitness < min then  min := fitness;   { nouveau min }
  end;
  { calculer la moyenne }
  avg := sumfitness / popsize;
end;
```

La procédure statistique *statistics* calcule les valeurs d'adaptation moyenne, maximale et minimale; elle calcule aussi *sumfitness*, qui est nécessaire pour la roue de loterie. Cette version de statistics est, une fois de plus, une solution minimale mais acceptable. On peut faire d'autres statistiques qui porraient

aussi être intéressants pour étudier en détail le parcours de l'algorithme après coup. La séparation des différentes fonctions statistiques dans la procédure *statistics* rend possible l'ajout des autres calculs.

Enfin, nous allons décrire le programme principal :

```
            { programme principal }
    begin
      gen := 0   { l'installation des structures }
      initialize;
      repeat                  { la boucle principale }
        gen := gen + 1;
        generation;
        statistics( popsize, max, avg, min, sumfitness, newpop);
        report( gen);
        oldpop := newpop;     { avancer d'une génération}
      until ( gen > maxgen )
    end.
```

Au début du code, on commence en initialisant le compteur de génération à 0, *gen* := 0. Puis on lit les données d'entrée- on ne le voit pas ici, car on n'a pas décrit toutes les procédures-, en initialisant une population aléatoire, et on continue en calculant des statistiques sur cette population initiale, et en afiichant un rapport initial spécial grâce à la procédure *initialize*. Enfin, une fois les préliminaires nécessaires effectués, on s'attaque à la boucle principale contenue dans le bloc repeat-until. Rapidement, on incrémente le compteur de génération, on engendre une nouvelle génération dans *generation*, on calcule les statistiques sur la nouvelle génération dans *statistics*, on affiche le rapport de génération dans *report* et on propage la population d'un seul coup :

```
oldpop := newpop;
```

Tout cela continue, pas après pas, jusqu'à ce que le compteur de génération dépasse le maximum, ce qui entraîne l'arrêt de l'exécution.

2.4. Aperçu des résultats théoriques relatifs aux AG

On peut dire que ces résultats sont d'un intérêt limité car, en pratique, on est rarement dans les conditions d'application des théorèmes. Néanmoins, leur message est transposable à des cas plus généraux et nous les citons parce qu'ils permettent d'éclairer le fonctionnement des AG sous un autre jour. Nous supposerons donc, dans ce paragraphe, que l'algorithme utilisé est l'Algorithme Génétique de base (SGA), manipulant des chromosomes binaires. Les théorèmes seront donnés sans démonstration (on peut les trouver dans la référence [GOL 91]).

Un des concepts centraux développés dans l'analyse théoriques des AG est le concept de "Schéma" décrit par HOLLAND. Il est fortement lié à la notion de" bloc de construction". Pour comprendre cette notion, nous devons adopter le point de vue suivant : pour guider sa recherche, un AG qui agit sur une population traite en fait des entités appelées "schémas" et tente de découvrir les meilleurs schémas et de les combiner. Un schéma est un masque de similarité reprenant un sous-ensemble de chromosomes possibles ayant des caractéristiques communes à certaines positions; c'est donc une hyper-surface particulière de l'espace de recherche. L'ensemble de tous les schémas couvre l'entièreté de l'espace de recherche avec une redondance élevée. Si l'alphabet des allèles est binaire (ce qui est le cas pour les SGA), un schéma est une chaîne constituée de 3 symboles possibles : 1,0 et * (symbole d'indifférence). Par exemple, le schéma 1***1 désigne tous les chromosomes qui commencent et finissent par 1. Le chromosome 10001 en est un représentant particulier; il est facile de voir que tout chromosome de longueur l est représentant de 2^l schémas.

Théorème1 : **Le nombre de schémas utiles traités par l'algorithme Génétique est de l'ordre n^3, si la population contient n individus**.

Ce théorème est aussi appelé "**théorème du parallélisme implicite**", parce qu'il exprime que l'AG qui travaille explicitement en parallèle sur n individus, manipule parallèlement et implicitement n^3 schémas. Quelques mots d'explication concernant ce parallélisme. Les algorithmes génétiques exploitent les meilleures régions de l'espace des solutions, parce que la répétition des reproductions avec le crossover augmente progressivement le nombre de chaînes dans ces régions : comme seules les meilleurs chaînes se reproduisent, elles se retrouvent avec leurs descendants dans les bonnes régions, ou régions cibles.

Le nombre de chaînes dans une région particulière croît à un rythme proportionnel à la qualité moyenne des chaînes de la région. Pour estimer la qualité moyenne de chaque région, un statisticien devrait considérer des dizaines d'échantillons issus de milliers ou de millions de régions : un algorithme génétique obtient le même résultat avec moins de chaînes et presque sans calcul.

Ce résultat surprenant découle de ce qu'une chaîne appartient à toutes les régions que ses bits définissent : la chaîne 11011001 appartient aux régions 11******, 1******1, **0**00*, etc. Les plus grandes régions, dont la définition comporte beaucoup de bits non spécifiés, sont généralement échantillonnées par une grande partie de l'ensemble des chaînes d'une population. Ainsi un algorithme génétique qui manipule une population de quelques milliers de chaînes échantillonne en réalité un nombre bien supérieur de régions. Ce "parallélisme implicite" confère aux AG leur principal avantage par rapport à d'autres méthodes de résolution de problèmes.

Le crossover complique les effets du parallélisme implicite. En algorithmique génétique, il sert à explorer de nouvelles parties des régions cibles, plutôt que d'exploiter les mêmes régions indéfiniment. Cependant il peut aussi "déplacer" un descendant d'une région à une autre, de sorte que le nombre de chaînes présentes dans les diverses régions ne soit plus strictement proportionnel à leur qualité moyenne. Cet effet ralentit la vitesse d'évolution. La probabilité qu'un descendant de deux chaînes quitte une région à laquelle appartiennent ses parents dépend de la distance entre les 1 et les 0 qui définissent cette région. Un descendant de deux chaînes de la région 10****, par exemple, n'en sort que si le crossover commence à la deuxième position, c'est-à-dire dans un cas sur cinq pour des chaînes à six bits (ou dans un cas sur 999 pour des chaînes de 1000 bits). En revanche, les descendants d'une chaîne à six bits exploitant la région 1****1 courent le même risque de quitter cette région quelle que soit la position du point de crossover. Les groupes de 1 et de 0 adjacents, dans la définition des régions, forment des "blocs de construction compacts" (ou blocs de construction ; ce sont, en fait, des séries de bits particuliers qui confèrent un avantage aux chaînes qui les portent, et par conséquent, leur proportion augmente progressivement pendant la reproduction) qui résistent bien au crossover et se perpétuent dans les générations futures avec une fréquence proportionnelle à la qualité moyenne des chaînes qui les contiennent. Si une reproduction avec crossover n'échantillonne pas toutes les régions proportionnellement à leur adéquation- ou adaptation- moyenne, elle échantillonne des blocs de construction compacts. Comme le nombre de ces blocs compacts est bien supérieur au nombre de chaînes d'une population (une chaîne peut contenir plusieurs blocs), les AG conservent leur parallélisme implicite malgré le crossover.

Comme en génétique biologique, deux gènes très éloignés chez les parents se retrouvent parfois proches l'un de l'autre chez leur descendants. Cette opération d'inversion permet l'établissement de blocs de construction plus compacts, résistant mieux au crossover que les blocs parentaux. Quand le nouveau bloc caractérise une région de forte adéquation, il s'impose par rapport aux blocs moins compacts, parce qu'il résiste mieux au crossover. Autrement dit, un système adaptatif utilisant l'inversion peut mettre au jour et favoriser des versions compactes de blocs de construction performants.

Le **parallélisme implicite** des AG leur permet de tester un grand nombre de régions de l'espace tout en ne manipulant que peu de chaînes. Il aide aussi les AG à affronter les problèmes non linéaires, où la qualité d'une chaîne contenant deux blocs de construction particuliers peut être bien supérieur (ou bien inférieur) à la somme des qualités individuelles de chaque bloc. Pour les problèmes linéaires, l'espace de recherche est réduit, parce que la qualité d'une chaîne due à la présence d'un 1 ou d'un 0 à une position particulière de la chaîne ne dépend pas de la présence d'un 1 ou d'un 0 quelque part ailleurs. Alors que l'espace des chaînes à 1000 bits contient plus de 3^{1000} (car chaque position peut contenir 0,1 ou *) chaînes, la résolution d'un problème linéaire ne nécessitera que l'examen des chaînes contenant un 1 ou un 0 à chaque position, soit 2000 possibilités. Lorsque le problème n'est pas linéaire, la difficulté augmente considérablement. La qualité des chaînes de la région *01***, par exemple, pourrait être supérieure à la moyenne de la population, même si la qualité des chaînes des régions *0**** et **1*** était inférieure. La non-linéarité n'implique pas l'inexistance de blocs de construction performants, mais seulement l'inutilité des blocs définis par un seul 1 ou par un seul 0. Cette caractéristique provoque une explosion des possibilités : l'ensemble de toutes les chaînes à 20 bits contient plus de trois milliards de blocs de construction. Par chance, le parallélisme implicite subsiste. Dans une population de quelques milliers de chaînes, beaucoup de blocs de construction compacts apparaîtront dans 100 chaînes ou plus, ce qui assure un bon échantillonage. Des blocs caractéristiques qui exploitent les non-linéarités pour obtenir des performances au-dessus de la moyenne sont automatiquement propagés dans les générations futures.

Définissons la qualité d'adéquation d'un schéma à la génération t, $f_s(t)$, par la moyenne des valeurs de la fonction de ses représentants dans la population à la génération t. Soit $f_m(t)$, la moyenne des valeurs de la fonction des individus constituant la population en t. De génération en génération, la distribution des schémas évoluent : ceux qui ont des représentants au-dessus de la moyenne vont prendre le dessus. L'évolution de la distribution est donnée par théorème 2.

Théorème2 **(théorème du schéma) : L'évolution du nombre de représentants n_s d'un schéma s est donnée par l'équation :**

$n_s(t+1) = (f_s(t)/ f_m(t))*(1-\varepsilon)* n_s(t)$ où ε est un nombre positif très petit.

En d'autres termes, les schémas de qualité d'adéquation au-dessus de la moyenne auront un nombre de représentants croissants exponentiellement. Le nombre ε modère la croissance en tenant compte du pouvoir "destructeur" de la mutation et du croisement; on montre que ε dépend de la spécificité du schéma (nombre de positions fixes) et de sa longueur (distance entre le premier et le dernier symbole fixe du schéma) ; ε est d'autant plus petit (et donc la croissance d'autant plus grande) que le schéma est peu spécifique et court ; un schéma possédant ces 2 propriétés est appelé "bloc de construction" dans la théorie mathématique des AG de base (SGA).

En pratique, le traitement implicite des schémas repose sur des estimées de la qualité d'adéquation f_s, elles-mêmes basées sur un nombre fini d'échantillons ; ces estimées sont soumises inévitablement à des erreurs d'échantillonnage, notamment par le fait tout simple qu'un individu aura forcément un nombre entier de descendants ; vu la nature itérative des AG, ces erreurs vont s'accumuler et donner des "trajectoires de recherche" bien différentes de celle que prédit la théorie si l'on ne prend pas de précautions spéciales. Concrètement, cela peut se manifester par une perte prématurée de diversité dans la population et par la convergence vers une solution sous-optimale. La propriété d'attribution exponentielle d'essais à un schéma est une propriété intéressante : on montre qu'en effet c'est la stratégie optimale de recherche dans un milieu stochastique inconnu (théorie de la décision).

On peut faire 2 reproches majeurs à ce théorème : d'une part, il n'est valable que dans la phase initiale de la recherche, quand les schémas sont encore distribués plus ou moins uniformément dans l'espace de recherche ; d'autre part, il ne considère que les effets de la sélection et considère le crossover

uniquement dans son aspect négatif (destruction de représentants du schéma). Comme la puissance motrice des AG naît de l'action combinée de la sélection et de la recombinaison par crossover, ce théorème, partiellement négatif, n'explique qu'une partie du fonctionnement.

D'un point de vue théorique, citons encore les développements suivants:

1. Caractérisation des problèmes dures pour les AG à partir d'une analyse en transformée de Walsh (la transformée de walsh est l'analogue de la transformée de Fourier dans le cas des fonctions définies sur des chaînes booléennes) [GOL 91]. En fait, l'idée essentielle est que l'AG risque d'échouer quand les bons schémas de faible spécificité ne contiennent pas les bons schémas de grande spécificité; à ce moment, la recherche initiale, se basant sur l'observation de la qualité d'adéquation des schémas de faible spécificité, guidera vraisemblablement mal les étapes futures.
2. Introduction de nouveaux opérateurs et amélioration des anciens pour une meilleure recombinaison des blocs de construction .
3. Introduction de nouveaux opérateurs et d'autres représentations afin d'introduire de la connaissance relative au problème.
4. Preuve de convergence vers l'optimum global (études sur la base d'une formulation en chaînes de Markov).

2.5. Les points constitutifs essentiels des AG

Ce qui fait l'attrait des AG est leur capacité d'accumuler de l'information sur un espace de recherche initialement inconnu et d'exploiter cette information pour guider la recherche future dans des sous-espaces prometteurs. Le mécanisme sous-jacents fondamental est la recherche et la combinaison de "blocs de construction" découverts durant les essais précédents ; cette notion est à mettre en rapport avec le concept de groupe co-adaptés d'allèles et de "*légo*" ; en particulier, les interactions non-linéaires entre gènes (phénomène appelé "épistasis" en biologie) font qu'une recherche uniquement basée sur la mutation ne serait pas efficace, alors qu'en procédant par échange et recombinaison la recherche peut se faire de façon plus efficiente (utilité du *sexe* en biologie). Ne perdons pas de vue que ce mécanisme est implicite, autrement dit qu'il n'y a pas de programmation explicite de ce mécanisme, mais qu'il émerge des propriétés d'auto-organisation induite par sélectionnisme. Outre ce mécanisme fondamental, quatre caractéristiques importantes coexistent dans un AG et distinguent celui-ci de nombreuses méthodes traditionnelles:

a) ***Parallélisme*** :
Les AG travaillent en parallèle sur certain nombre de points-candidats (solutions potentielles), et non pas sur un candidat unique; le parallélisme est essentiel pour le mécanisme de recombinaison (si l'on ne travaillait que sur seul point, on ne pourrait évidemment pas définir de recombinaison) ; n'oublions pas que la méthode de recherche n'est pas locale mais globale et distribuée sur tout l'espace de recherche, en un certain sens (exprimé par le théorème du schéma), la population constitue une base de données compacte (mémoire) qui résume toute l'information acquise par la recherche jusqu'à la génération considérée ; le traitement parallèle est aussi très attrayant dans l'optique d'une mise en oeuvre informatique sur machine parallèle;

b) ***Manipulation d'entités arbitraires*** :
Les AG manipulent des entités qui ne sont pas forcément numériques; en fait, un AG peut travailler sur n'importe quel espace de recherche, à condition que les points de cet espace de recherche soient toujours constitués d'un ensemble d'entités élémentaires (les gènes ou les caractéristiques) sur lesquelles il est possible de définir des opérateurs de mutation et de crossover ;

c) ***Utilisation minimale d'information a priori :***
Les AG ne requiert de l'environnement qu'une mesure de l'adéquation (ou de la qualité) d'un individu; il ne repose sur aucune autre information (par exemple des dérivées), ni hypothèse telles que la continuité et la différentiabilité de la fonction d'adéquation; en fait, certaine versions des AG (avec

sélection par rangement) ne requiert de l'environnement qu'une capacité à classer les individus entre eux, et rien d'autre;

d) ***Balance entre exploration et exploitation***

Les phases de sélection et de reproduction sont exécutées en utilisant des règles probabilistes plutôt que des règles déterministes; l'introduction du hasard a, entre autres buts, celui de maintenir les propriétés d'exploration lors de la recherche; cela est non seulement bénéfique pour l'optimisation de fonctions multi-modales (présentant plusieurs optima), mais aussi en cas de fonction non-permanente (déplacement ou changement des optima au cours du temps), où le caractère adaptatif d'un algorithme prend de l'importance; en outre, certaines versions des AG (les SGA, avec sélection proportionnelle) réalisent automatiquement une distribution presque optimale des essais, ce qui signifie que les AG gèrent le compromis exploration/exploitation d'une façon presque optimale ;

Nous définirons donc un Algorithme Génétique comme un algorithme qui possède ces 4 caractéristiques en plus de l'heuristique de recherche réalisant la "combinaison de blocs de construction" par l'action simultanée de la sélection et d'un mécanisme de recombinaison (crossover).

Ces 4 caractéristiques ne sont évidemment pas l'apanage des AG ; d'autres méthodes possèdent, au moins partiellement, ces propriétés: par exemple, le "recuit simulé" (propriétés b, c et d), le "simplex" (propriétés a et c), tout "grimpeur stochastique parallèle" (la méthode ne porte sans doute pas de nom connu, mais elle est facilement imaginable : il suffit de lancer en parallèle plusieurs grimpeurs en ajoutant un peu de hasard dans leur déplacement afin de maintenir un niveau minimum d'exploration, et de prendre le meilleur résultat; cette méthode posséderait au moins les propriétés a, c et d). Le mécanisme sous-jacent de "combinaison de blocs de construction" est souvent considéré comme original et propre aux AG, mais on pourrait sans trop de difficulté prétendre, à juste titre, que le "simplex" utilise aussi une certaine forme de combinaison (les (n+1) individus constituant le simplex se "reproduisent" pour donner un nouvel individu par réflexion, expansion ou contraction).

Dans les AG classiques, il existe encore un trait distinctif supplémentaire: les AG nécessitent le codage des paramètres à optimiser et opèrent sur le code plutôt que sue les paramètres eux-mêmes (cf. Annexe 4.5 « Distinction génotype/phénotype en génétique »). Le rôle du codage est double : d'une part, la transformation du problème d'optimisation original en un problème combinatoire puisque les AG sont essentiellement des méthodes de recherche des "legos" et de l'auto-organisation) et plus indépendante du problème ; c'est ensuite sur la représentation bas-niveau que des opérateurs tout à fait généraux vont pouvoir agir pour faire évoluer le système. Par exemple, dans le cas des problèmes d'optimisation numérique continue (où les paramètres à rechercher sont des nombres réels), un codage fréquent est le codage binaire[3] du nombre en une chaîne finie de 0 et 1. De cette façon, les problèmes continus sont traités comme des problèmes discrets, avec une perte de précision inévitable; c'est pourquoi il faut veiller à une discrétisation correcte de l'espace de recherche (un peu comme en théorie de l'échantillonnage). Il ne faut pas oublier non plus qu'un codage introduit des liens et des propriétés dans la nouvelle représentation qui n'étaient pas présents dans la représentation originale; de même, certaines propriétés de la première représentation se perdent dans le codage.

De toute façon, le problème de la représentation est problème crucial lors de l'application des AG . Il faut prendre soin de trouver une représentation qui permette la recherche, l'isolation (implicite) et la recombinaison de blocs de construction. En effet, le bon fonctionnement de l'opérateur fondamental (recombinaison implique un comportement particulier des parties en coopération: il faut que la combinaison de 2 "bons legos" puisse conduire effectivement vers une meilleure solution et ne donne

[3] Si l'on veut conserver la typologie de l'espace de départ, il vaut évidemment mieux adopter le codage binaire Gray; de cette façon, une mutation d'un bit ne peut jamais donner lieu à un déplacement immense dans l'espace des paramètres de départ: 2 solutions proches dans l'espace codé correspondent à 2 points proches dans l'espace non-codé.

pas lieu systématiquement à une solution détériorée (hypothèse que nous appellerons *condition de reconstructibilité par recombinaison*) ; notons que le cas linéaire vérifie trivialement cette condition, mais à ce moment-là il existe une méthode d'optimisation appropriée, autrement plus performante que les AG; dans les cas non-linéaires, il existe une formulation basée sur les coefficients de Walsh de la fonction [GOL 91] (généralisation des coefficients de Fourier pour les fonctions définies sur des vecteurs booléens) qui permet de voir si la condition est vérifiée, mais elle est d'un intérêt pratique limité car elle nécessite la connaissance de la valeur de la fonction à optimiser en chaque point de l'espace de recherche. Du point de vue du relief de la fonction à optimiser, cette condition revient à dire que les optima de la fonction ne sont pas situés n'importe où, que la position de l'extremum global (cette interprétation est évidente quand se raccroche à la notion de "combinaison") et qu'en outre, la région située entre 2 bons optima locaux est une bonne région de recherche pour des optima locaux encore meilleurs. Il arrive qu'en transformant un relief (et donc la fonction) qui n'obéirait pas à cette condition, par l'application d'une transformation relativement facile à trouver, la condition soit alors satisfaite.

Profitons du moment pour émettre une seconde remarque: l'AG n'exploite pas assez les informations locales : pourquoi ne pas tenir compte des leçons apprises dans le voisinage ? A cette question Goldberg [GOL 91] suggère l'utilisation des AG guidés par la connaissance C'est l'utilisation des différents moyens afin de combiner l'information spécifique à un problème avec les AG. Parmi ces moyens on trouve des techniques hybrides (hybridation entre les AG et les méthodes de type "grimpeurs"), des opérateurs dirigés par de la connaissance et des méthodes d'approximation de fonctions. Examiner en détail chacune d'elle nous emmènerait au-delà de l'objectif de ce travail (le lecteur intéressé trouvera plus d'information dans la référence [GOL 91]).

2.6. Applications

2.6.1 Introduction

En matière d'applications des AG, le champ est très fertile, et il ne passe pas un mois sans lire un article sur telle application ou tels résultats de recherche sur les AG. Vu le scope limité de mon travail, je me contente de citer deux articles récents, puis d'énumérer, mais pas exhaustivement, des applications fortement connues et utilisées, et pour finir je décrirai avec un peu plus de détails, l'intégration des AG dans les réseaux de neurones.

Commencons d'abord par l'article qui est paru dans le magazine ("La Recherche" -novembre 2000, se référant elle-même à *H. Lipson et J.B. Pollac, Nature, 406,974, 2000* .") sous le titre: "robots darwiniens", où on peut lire le suivant :

« Peut-on imaginer un robot qui serait, de sa conception à sa réalisation et sa fabrication, entièrement piloté par une intelligence artificielle sans l'intervention de l'homme ? Deux chercheurs américains y sont parvenus en élaborant un logiciel capable de simuler l'évolution de modèles de robots faits de tubulures et d'articulations. Partant d'une population de 200 robots virtuels et vierges de tout "génome", la simulation modifie pour chacun d'eux des éléments et détermine leur aptitude à se mouvoir dans un environnement prédéterminé. La descendance qui voit le jour est rajoutée et à nouveau sélectionnée. L'opération est renouvelée de 300 à 600 fois. Les robots virtuels finalement sélectionnés sont élaborés en 3 D et fabriqués directement, par une technique de prototypage rapide, en polymère plastique : il ne reste qu'à y ajouter à la main des petits moteurs. Aucun logiciel n'étant transféré sur les robots une fois construits, il n'y a donc pas de rétroaction du monde physique dans le processus d'évolution. ».

Le deuxième article est paru dans Scientific Americain - January 2001, sous le titre « Complexity theory_software - Complexity's Business Model ». Celui-ci parle plutôt des applications - avec des résultats très significatifs- des AG dans le domaine, si vaste et si juteux, de Business, en particulier,

dans les problèmes difficiles d'ordonnancement et d'optimisation. Dans cet article, on peut lire que la plupart des software de la compagnie **i2 Technologies**, Irving, Tex. - un producteur leader dans le software de e-commerce - utilisent les algorithmes génétiques pour optimiser les modèles de l'ordonnancement de production.

Des centaines de milliers de détails, inclus dedans les commandes des clients , les disponibilités de matériels et de ressources, les capacités de production et de distribution, et les dates de livraisons, sont tous incorporés dans le système. Puis les AG introduisent " mutations" et "crossovers" ne vue générer les candidats d'ordonnancements qui seront évalués par une fonction d'aptitude - fitness-, explique le consultants stratégiques de i2 , Gilbert P. Syswerda, un cofondateur d'Optimax - une start-up dans le design de software de scheduling -. " Les algorithmes génétiques ont montré leur importance en générant des nouvelles solutions dans beaucoup de domaines " dit J. HOLLAND (c'est lui qui a élaboré et jeté les bases des AG en 1975). Il n'existe aucune contre-partie(ou équivalent) à ce type de métissage ou croisement dans les analyses traditionnelles d'optimisation ajoute-t-il.

La société **International Truck and Engine** (précédemment **Navistar**), par exemple, a récemment installé le software de i2. En introduisant l'ordonnancement adapté aux changements, le software a effectivement débarrassé la production des difficultés imprévues qui peuvent glisser dans la chaîne de l'approvisionnement et faire craindre un profond dérèglement. En fait le software a baissé de 90% les perturbations très coûteuses dans cinq sites de International Truck, selon Kurt Satter, le manger du système de transport Goliath. Le software d'optimisation basé sur les algorithmes génétiques peut aussi trouver les points d'étranglement ,et prévoir les effets, des changement de la ligne de production ,des introductions de nouveaux produits et même des compagnes publicitaires, affirme Syswerda. Les milliers des contraintes sous lesquelles les businesses fonctionnent, peuvent aussi être encodées. Telle modélisation non-linéaire est fondamentalement et totalement impossible avec des outils de programmation conventionnelle, soutient-il. Ainsi, on voit avec quel type de problèmes on peut utiliser les algorithmes génétiques, et à quel degré de succès peut-on s'attendre.

Quant à l'énumération - non exhaustive- des autres applications des AG, on a l'embarras du choix. Citons par exemple: Reconnaissance vocale - Design des réseaux en télécommunications - Intégration des AG et de la Logique Floue - Imagerie médicale - Conception d'une turbine de réacteur d'avion - Réseau de pipelines qui transporte le gaz - ...etc.

Enfin, on va décrire la combinaison des AG avec les RN (Réseaux de Neurones, ou Réseaux Neuronaux ; en anglais NN ou Neural Networks).

2.6.2. Les systèmes de classeurs

Les systèmes de classeurs constituent une extension importante des AG, proposée et élaborée par J. Holland à partir des années 1980. Ouverts sur l'environnement par l'intermédiaire de "senseurs" et "d'effecteurs", ils représentent explicitement une structure cognitive équipée d'une mémoire à court terme et d'une mémoire à long terme. La mémoire à long terme est constituée de règles de production, codées sous la forme de chaînes de caractères susceptibles d'évoluer sous l'effet d'un AG. L'importance fonctionnelle de ces chaînes dépend de leur aptitude passée à engendrer des récompenses et à éviter des punitions. C'est pourquoi les systèmes de classeurs servent à étudier et résoudre les problèmes d'apprentissage par machine.

Décrivons un peu plus précisément ces systèmes. Un "système de classeurs" est un ensemble de règles (nommées classeurs) qui vérifiées. Les conditions et les actions sont représentées par des chaînes de bits correspondant à la présence ou à l'absence de caractéristiques particulières dans l'entrée ou la sortie des

règles ; pour chaque caractéristique présente ; la chaîne contient un 1 à une position appropriée, un 0 pour chaque caractéristique absente.

Une règle de reconnaissance de chiens, par exemple, serait codée par une chaîne contenant des 1 aux places correspondant) "poilu", "bave", "aboie", "loyal", "court après les balles", et des 0 aux places correspondant à "métallique", "parle l'ourdou" ou "possède une carte bancaire". En pratique, le programmeur doit choisir les caractéristiques les plus simples et les plus fondamentales afin que, comme dans le jeu des 20 questions, on puisse classer les objets et les situations les plus variés.

Parfaites pour la reconnaissance, ces règles déclenchent en outre des actions si l'on a établi une correspondance entre leurs bits de sortie et des comportements; par exemple, un système de classeurs qui simulerait une grenouille devrait contenir des chaînes qui réagiraient aux objets vus par la grenouille. Selon la taille, la position et diverses autres particularités des objet perçus, la grenouille les attaquerait, les poursuivrait ou les ignorerait.

Tout programme écrit dans un langage de programmation standard, tel le FORTRAN, le BASIC ou le LISP, peut être réécrit sous la forme d'un système de classeurs. Pour faire évoluer des règles de classeurs en vue de résoudre un problème particulier, on part d'une population de chaînes aléatoires et 1 et de 0, et on classe chaque chaîne d'après son adaptation, c'est-à-dire la qualité de sa résolution du problème. Selon la nature de ce dernier, la mesure de l'adaptation peut être un rendement, un gain à un jeu ou une proportion d'erreurs, par exemple. En propageant par reproduction les chaînes de qualité et en éliminant, la proportion des chaînes adaptées. Comme la reproduction engendre sans cesse de nouvelles chaînes, des solutions de plus en plus performantes apparaissent. La stratégie s'applique à des problèmes variés, de la théorie des jeux à la conception de systèmes mécaniques complexes.

Pour résumer, on peut dire que pour construire un algorithme informatique capable d'évoluer, il faut trouver un moyen de représenter les programmes de façon qu'un changement du génotype (les bits qui constituent le programme) provoque un changement approprié du phénotype (ce qui fait le programme) ; un système de classeur peut parfaitement remplir cet objectif car il est composé de chaînes de bits (nommées classeurs) ; les bits représentent les caractéristiques possibles des entrées du programme et de ses actions.

2.6.3. Applications des AG en Commande de processus

Nous présenterons ici une application des AG en commande de processus. Les AG peuvent être utilisés en tant que méthode de recherche combinatoire incluant, outre des des propriétés basées sur le parallélisme et l'exploration, des heuristiques de recherche intéressantes basées sur des principes d'auto-organisation. Comme un grand nombre de problème de commande, en ligne ou hors ligne, requièrent des méthodes de recherche, les AG peuvent constituer une alternative intéressante lorsque les méthodes d'optimisation traditionelles (descente de gradient, méthode "grimpeur", méthodes analytiques telles que les moindres-carrés) ne parviennent pas à fournir efficacement des résultats fiables.

Le terme "commande de processus" doit être compris dans son acceptation la plus large, incluant toutes les étapes "automatisables" entre les objectifs économiques de production (marché et gestion) et les mécanismes d'asservissement des organes à la base de la production.

2.6.4. Application à la recherche d'automates à éléments finis (problème d'identification)

- **Le problème**: Modélisation par automates à éléments finis

Une application directe des AG à la commande en ligne et, si ,possible, adaptative de processus est rarement possible (le problème de convergence des AG rend toute approche directe d'AGs en ligne est rarement faisable en pratique, particulièrement en milieu industriel, où les contraintes de production laissent peu de place aux "divagations" temporaires des AG. Cette difficulté d'emploi a poussé les chercheurs à imaginer d'autres modes d'application, où les AG agissent soit indirectement soit partiellement sur le processus à commander.L'une d'entre elles, celle consistant à scinder le problème de la commande en deux, semble être très adéquate et prometteuse. Le rôle des AG est alors de découvrir par ses propres mécanismes, ou en combinaison avec d'autres techniques, un modèle satisfaisant du processus ou d'autres informations manquantes (par exemple, un bon estimateur d'état).

Pour montrer l'efficacité des AG dans la construction automatique de modèle, nous considérons le problème suivant: supposons que la dynamique du processus à modéliser soit telle qu'il est possible de partitionner l'espace d'état en nombre fini de sous-régions à travers lesquelles le système se déplace de façon déterministe en fonction des sigaux d'entrée[4]. Le comportement du processus peut alors être décrit au moyen de règles de transition déterminant l'évolution d'un automate à éléments finis.

Dans ce formalisme, chaque noeud du diagramme d'état de l'automate représente une sous-région particulière de l'espace d'état. Les signaux d'entrée et de sortie doivent aussi être discrétisés pour pouvoir être représentés par les symboles d'un alphabet fini. A chaque état de l'automate, on associe un signal de sortie (ou plusieurs, si le processus possède plus d'une sortie); ce signal de sortie est la seule information disponible (mesurée), mis à part le signal d'entrée. L'automate fonctionne de façon synchrone, en ce sens qu'une et une seule transition est effectuée à chaque instant k. Les équations dynamique du système sont donc:

$$x_{k+1} = f(x_k, u_k)$$
$$y_k = g(x_k)$$

où x_k prend n valeurs discrètes possibles. x_{k+1} représente l'état (ou plus exactement une sous-région particulière de l'epace d'état) du système à l'instant k+1 qui est fonction de l'état "passé" x_k du système, et du signal d'entrée –binaire- u_k, tous les deux à l'instant précédent – ou "passé" – k. y_k représente le signal de sortie – on a supposé qu'il y a une seule sortie – associé à l'état x_k, et donc c'est une fonction de cet état.

Le problème de modélisation devient alors celui de la recherche d'un automate à éléments finis ou,de façon équivalente, celui de l'identification des fonctions f et g dans l'équation d'état ci-dessus. Pour découvrir l'automate qui décrit le plus adéquatement possible le comportement d'un processus dans un environnement donné, il est nécessaire de concevoir un algorithme qui opérera sur les séquences d'entrée et de sortie observées en considérant le processus comme une "boîte noire" d'état initial inconnu. Cet algorithme devra apprendre graduellement à prédire le symbole de sortie futur, ce qui revient à la découverte d'un automate adéquat.

Les AG constituent un moyen pour atteindre cet objectif. Fondamentalement, les AG vont maintenir une population de candidats-automates et exploiter leurs prédictions pour former de nouveaux candidats avec des performances améliorées; dans ce cas, la fonction d'adéquation est définie par une certaine mesure de la qualité de prédiction. Bien sûr, il faudra s'assurer que le mécanisme de codage d'un automate (représentation sous forme "chromosomique") respecte la condition de *reconstructibilité par recombinaison* (cf. §2.4.).

- **Exemple et résultats**

[4] Les systèmes chaotiques ne jouissent certainement de cette propriété, car 2 points proches appartenant à la même sous-région auraient tôt fait de se mouvoir dans des régions différentes; un formalisme de type "automate à éléments finis déterministe" ne conviendrait donc pas.

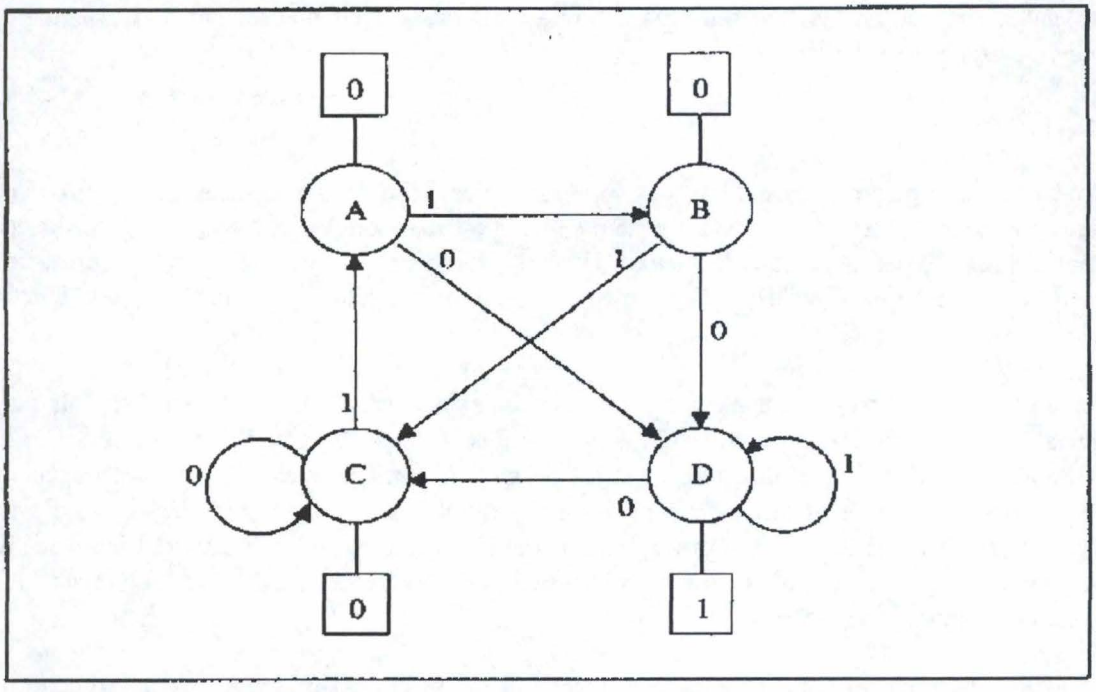
Envisageons à présent la découverte d'un automate à 4 états (donc n = 4), à entrée et sortie binaires. L'automate peut être spécifié par un tableau donnant, pour chaque état possible, la sortie associée ainsi que les états futurs pour chacune des valeurs possibles du signal d'entrée. Un exemple d'automate à 4 états est donné en la figure ci-après et le tableau de définition correspondant est représenté par le tableau suivant. Nous supposerons que le nombre d'états de l'automate (n) est connu.

Etat courant	Sortie	Etat suivant	
		Entrée = 0	Entrée = 1
A	0	D	B
B	0	D	C
C	0	C	A
D	1	C	D

Tableau de définition d'un automate à 4 états

0	D	B	0	D	C	0	C	A	1	C	D

Représentation chromosomique de l'automate de la figure précédente



Exemple d'automate à 4 états

Remarque 1:

Dans l'automate représenté dans la figure précédente, on n'a pas désigné un état initial, et des états acceptants ; cela par soucis de généralité car l'automate modélise le comportement (dynamique) d'un processus quelconque. De plus, on peut choisir n'importe quel état comme état initial, et n'importe quel(s) état(s) comme état(s) acceptant(s). En tout cas, cela ne changera rien dans le tableau de définition de l'automate.

Remarque 2:

Il y a une bijection entre l'ensemble des automates et celui des tableaux de définition. Par conséquent, il y a aussi une bijection entre les automates et les chromosomes (qui sont chacun l'enchaînement des lignes du tableau de définition correspondant). Chaque gène du chromosome représente une ligne, en respectant l'ordre : première ligne- premier gène, deuxième ligne-deuxième gène... Le but dans cette application est de trouver un automate avec une certaine configuration (des sorties et des états suivants bien précis, sachant que le premier état suivant dans le chromosome est par convention celui lorsque l'entrée égale 0 tandis que le deuxième état suivant est celui pour l'entrée égale à 1). Quand on est dans un état (ou quand on entre dans un état), l'état courant, cet état produit une sortie déterminée uniquement par cet état. De plus, la transition vers l'état suivant dépendra de la valeur de l'entrée ainsi que de l'état courant.

⇒ **Le codage**

Le procédé de codage adopté est très simple: à partir du tableau de définition d'un automate, on obtient directement le chromosome en "enchaînant" les différentes lignes du tableau et en attribuant un symbole arbitraire à chaque état possible de l'automate. Le choix de la représentation binaire des états est tout à fait possible, mais introduit des propriétés parasites inutiles et même gênantes. On remarquera que le chromosome ainsi obtenu (voir figure précédente) contient des gènes dont la valeur n'appartiennent pas toutes au même alphabet (les cases correspondant à une sortie doivent prendre une valeur binaire, tandis que les autres seront occupées par des symboles représentant les états possibles); il faut donc changer légèrement les AG traditionnels de façon à pouvoir manipuler de telles entités avec cohérence et consistance[5]. Notons aussi une difficulté inhérente au problème: la représentation n'est pas unique, car on peut permuter tous les symboles d'état et obtenir exactement le même comportement: il y a donc au moins *n!* solutions possibles équivalentes et les AG tomberont sur l'une ou l'autre en fonction de la population initiale; la tâche n'est donc pas si aisée pour les AG puisque, si certains individus parviennent à capturer diffférentes parties de la dynamique de l'automate, la combinaison de ces individus ne sera pas nécessairement performante si les représentations internes de ces individus diffèrent; néanmoins, le problème de multiplicité de représentation mis à part, la reconstructabilité par recombinaison semble être valable pour ce codage.

⇒ **La fonction d'adéquation(ou d'évaluation)**

Pour évaluer la fonction d'adéquation de chaque individu, on fournit à chaque génération un nombre fixé de séquences aléatoires de signaux d'entrée de longueur donnée, chaque séquence agissant sur un état initial aléatoire de l'automate. Chaque individu de la popualtion produit alors les séquences correspondantes de valeurs prédites de la sortie de l'automate; en réalité, chaque individu va proposer n alternatives, chaque alternative correspondant à la séquence de prédictions pour état intial donné.
Le score partiel d'un individu relatif à une séquence de signaux d'entrée est déterminé par le nombre de réponses correctes fournies par la meilleure des n séquences alternatives. Les scores partiels sont additionnés pour les diverses séquences d'entrée et normalisés afin de donner le score d'adéquation final. quand un individu a correctement prédit toutes les séquences d'entrée présentées à sa génération, on lui soumet aussi les séquences relatives au 10 générations précédentes, de manière à raffiner et améliorer le processus de sélection entre les meilleurs individus. Les détails techniques des mises en oeuvre adoptées sont les suivants:

* longueur des chromosomes: 20 (bits)
* taille de la population: 30
* probabilité de mutation: 0.006 (par bits)
* probabilité de croisement(croisement à 1 point): 0.6(par individu)

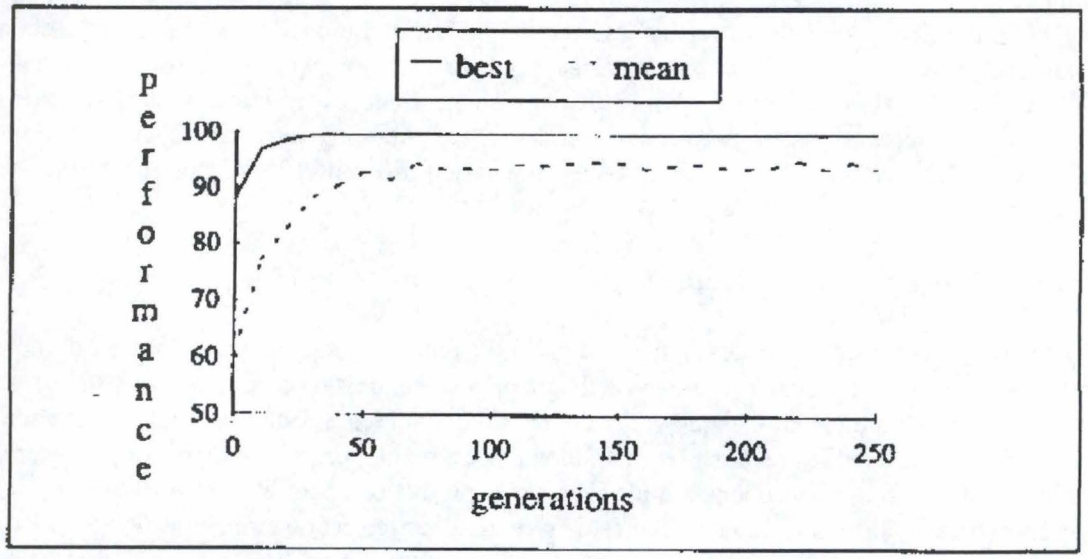
[5] En fait, seul l'opérateur de mutation doit être étendu au cas d'allèles non-binaires, appartenant à des alphabets variables.

* stratégie: élitiste
* sélection: par rangement

On notera que la fonction d'adéquation définie ci-dessus procure seulement une estimée de la qualité de prédiction d'un individu et ne donne pas directement une mesure de similarité (à cause que la représentation n'est pas unique) entre l'individu et l'automate recherché; en ce sens, la fonction peut être considérée comme "bruittée" puisque le même individu peut obtenir différents scores d'adéquation pour des séquences d'entrée différentes.

⇒ **Elaboration et résultats**

Donc, nous avons utilisé les AG pour la recherche de l'automate à 4 états décrits par le tableau précédent. La longueur d'un chromosome est donc de 12(20 bits auraient été nécessaires pour une représentation binaire; 4 pour les 4 bits de sortie, et 2 par lettre d'état, car on a 4 lettres=2^2). A chaque génération, on présente 10 séquences d'entrée avec des états initiaux différents, chaque séquence étant composée de 10 signaux d'entrée successifs. La figure suivante présente l'évolution des performances du meilleur individu et des performances moyennes (moyenne des performances de la population), toutes les valeurs étant moyennées sur 50 expériences avec des populations initiales différentes. On observera d'excellents résultats sont déjà atteints après 30 générations (de 30 individu). Néanmoins, il faut attendre quelques 200 générations pour produire l'automate correct avec une fiabilité quasi-absolue. Rappelons que cette solution correspond à l'un des 4! (=24) automates équivalents décrit le tableau précédent, parmi les 2^{20}($\cong 10^6$) automates possibles.



Evolution des performances lors de l'apprentissage(automate à 4 états)

Nous pouvons comparer les performanes des AG avec, par exemple, celles fournies par l'approche un peu brute de l'énumération systématique, en évaluant les efforts de calcul des 2 approches avant d'arriver à la bonne solution (en moyenne). La comparaison est donc basée sur le nombre moyen d'essais nécessaires pour trouver l'automate correct. Pour la procédure d'énumération, on peut montrer que ce nombre moyen est donné par $(2n^2)^n$ / (n!+1). Pour les AG, ce nombre est obtenu expérimentalement en multipliant le nombre moyen de génération avant d'arriver à l'automate correct par la taille de la population. Le tableau suivant présente les résultats de la comparaison entre les 2

approches pour les automates à 4, 6 et 8 états[6]. Comme on le remarquera, la supériorité des AG sur la "force brute" est très nette dès que le nombre d'états dépasse 4.

	Nombre d'essais		
	4-états	6-états	8-états
Enumération systématique	41943	1.9×10^8	1.79×10^{12}
Algorithmes Génétiques	6300	34000	140000

Comparaison du nombre moyen d'essais avant de trouver un bon automate

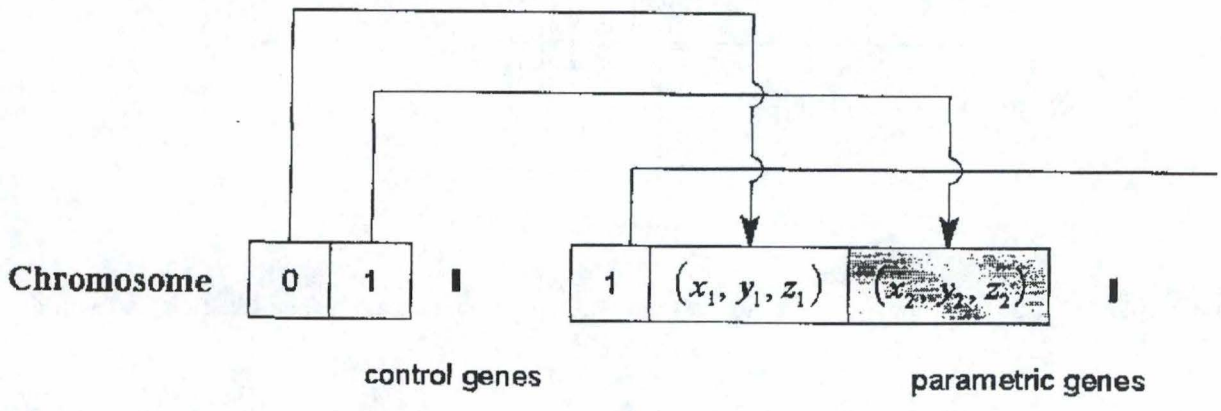
⇒ **Conclusions**

Sous certaines conditions, le problème de modélisation d'un processus peut se réduire à la recherche d'un automate à éléments finis. Nous avons montré expérimentalement que les AG pouvaient très bien se charger de cette tâche, que l'objectif était atteint avec une efficacité supérieur à la simple énumération systématique dès que le nombre d'états était supérieur à 4 et que cette supériorité allait croissante avec le nombre d'états. Même si le problème peut sembler artificiel au départ, la méthodologie reste toutefois valide pour les problèmes de modélisation de la dynamique d'un processus, lorsqu'il s'agit de découvrir automatiquement et de manière adaptative des règles de transition entre les sous-régions d'un espace d'état sans le besoin d'information *a priori*.

2.6.5. Le design d'un réseau de télécommunications en utilisant les AG

Il existe un autre domaine où l'utilisation des AG est très appropriée. Dans ce domaine, l'optimisation est souvent exigée pour l'ordonnancement (scheduling), la planification des ressources et pour le design de la topologie du réseau. Les AG ont déjà démontré leur puissance et leur efficacité pour résoudre les problèmes NP.

Les AG hiérarchiques HGA (Hierarchical GA) peuvent être utilisés pour le design des réseaux locaux sans fil (WLAN). La structure d'un chromosome de HGA est illustrée dans la figure suivante.



La structure du chromosome génétique hiérarchique pour le design du WLAN

[6] En réalité, le temps dépensé pour tester une solution avec l'énumération systématique est moindre qu'avec les AG; la raison en est qu'il suffit d'une prédiction incorrecte pour disqualifier un candidat, tandis qu'avec les AG il est nécessaire de poursuivre le processus d'évaluation sur toutes les séquences-tests afin de mesurer la qualité de prédictiond'un candidat de façon suffisamment précise. Mais, même en comptant que l'évaluation par AG prend 100 fois plus de temps, la supériorité des AG reste incontestable (voir tableau ci-dessus).

Les gènes de coefficients spécifient les endroits (ou leurs coordonnées) où se trouvent les stations. Une telle approche ne satisfait pas seulement l'optimisation des contraintes et des fonctions aux multi-objectifs compliqués, mais elle identifie aussi le nombre exact de stations requises. Cela a été démontré par un design d'un WLAN qu'on a réalisé dans un département de radiodiagnostique d'un hôpital [TAN97].

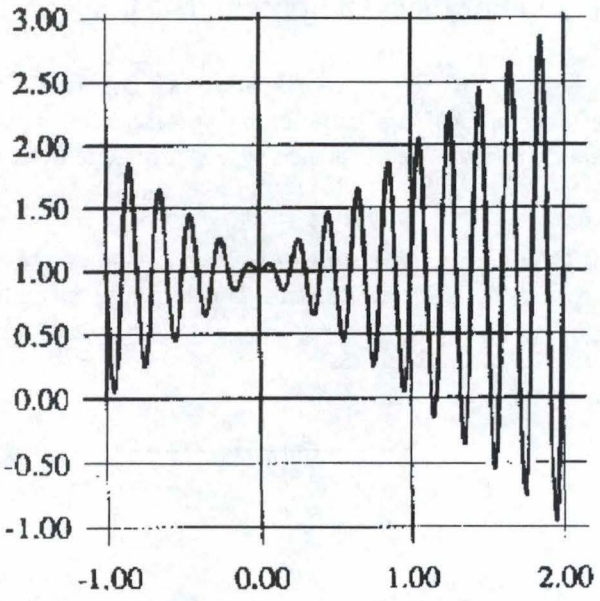
Par ailleurs, dans une autre étude [KTK97] on a proposé l'utilisation des AG dans une solution totale pour le design d'un réseau de communications à commutation par paquets (comme par exemple, un système téléphonique hiérarchique composé de centres régionaux,sectoriels,...,utilisateurs finaux). Avec les AG on a, pas seulement optimisé la topologie du réseau, mais on a aussi amélioré la capacité et le routage. On a reporté que les résultats obtenus étaient supérieurs même aux méthodes avec heuristiques.

2.6.6. L'optimisation d'une fonction simple

Nous allons étudier les différentes caractéristiques d'un AG utilisé pour l'optimisation d'une fonction simple d'une variable. La fonction est définie de cette façon :

$$f(x) = x * \sin(10\pi x) + 1.0$$

et elle est représentée dans la figure suivante:



Le problème est de trouver x qui appartient à l'intervalle [-1...2] qui maximise la fonction f, c'est-à-dire trouver x_0 tel que:

$$f(x_0) \geq f(x), \text{ pour tout } x \in [-1...2].$$

Il est relativement aisé d'analyser la fonction f. Les zéros de la première dérivée f' doivent être déterminés:

$$f'(x) = \sin(10\pi x) + (10\pi x) * \cos(10\pi x) = 0;$$

On divise les deux membres de cette équation par $\cos(10\pi x) \neq 0$,ce qui donne :

$$\tan(10\pi x) = -10\pi x.$$

Le cas $\cos(10\pi x) = 0$ est une solution de l'équation précédente, car si $\cos(10\pi x) = 0$, cela implique que $\sin(10\pi x) = 0 \Rightarrow x = 0$ ou $x = 2n.\pi$ (où n = 0,1,2,...),qui est une solution de l'équation précédente, donc les deux cas $\cos(10\pi x) = 0$ ou $\cos(10\pi x) \neq 0$ sont o.k.

L'équation : $\tan(10\pi x) = -10\pi x$ possède une infinité de solutions sur la droite des réels :

$x_i = (2i-1)/20 + \varepsilon_i$, pour i = 1,2,...

$x_0 = 0$

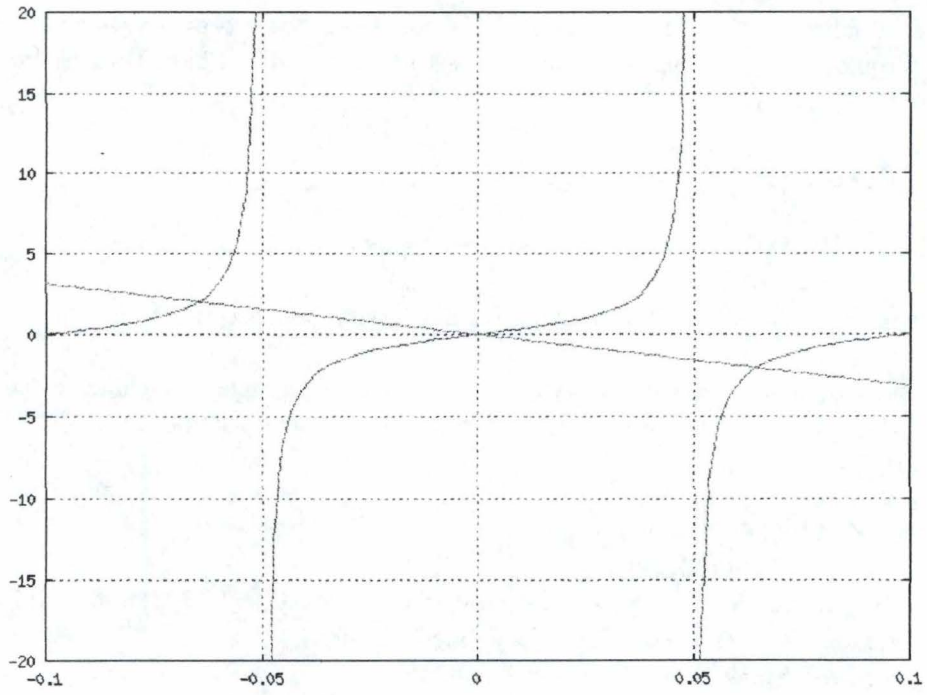
$x_i = (2i+1)/20 - \varepsilon_i$, pour i = -1,-2,...

Où les termes ε_i représentent une suite décroissante de nombres réels(pour i = 1,2,...,et i = -1,-2,...) qui s'approchent du zéro. Notons aussi que la fonction f atteint ses maxima locaux pour x_i si i est un entier pair(voir la figure).

Les solutions de l'équation (2) sont les points d'intersection des graphes des deux fonctions:
$y = \tan(10\pi x)$, et $y = -10\pi x$. Le premier graphe étant celui d'une droite passant par l'origine , et ayant la pente négative, donc elle est située dans le deuxième et quatrième quartier du plan. Le deuxième graphe est celui de la fonction tangente.
Vu que x = 0 est une solution de l'équation (2), et vu que *tan* est une fonction périodique, donc il prend périodiquement les mêmes valeurs (ici dans notre cas, la période de cette fonction est $\pi/10\pi = 1/10$), et par conséquent, les autres points d'intersection où les deux y sont égaux :

$$\tan(10\pi x) = \cos(10\pi x)$$

Doivent être, eux aussi, périodiques, et de période 1/10, ce qui est le cas, car on remarque que les distances, en valeur absolue, entre les x_i = 1/20, 3/20, 5/20, ...ou les x_i = -1/20, -3/20, -5/20, ... sont effectivement égales à 1/10.
Les deux premières solutions de (2) sont $x = \pm 1/20 \pm \varepsilon_i$,les deux deuxièmes se trouvent à une distance ± 1/10,donc $x = \pm 3/20 \pm \varepsilon_i$,...;Les asymptotes de $\tan(10\pi x)$ se trouvent à ±1/20, ±3/20,... (pour rappel, les asymptotes de tan(x) coupent l'axe X en $\pm\pi$, $\pm 2\pi$, $\pm 3\pi$, ...), et la droite $y = -10\pi x$, coupe ces asymptotes -de $\tan(10\pi x)$- en ±1/20, ±3/20, ...;comme on cherche les points d'intersection de la droite, non pas avec les asymptotes, mais plutôt avec le graphe de la tangente, on a du ajouter ou soustraire des petites quantités décroissantes et qui tendent vers le zéro, car en partie positive de l'axe X le graphe de la tangente se trouve à droite (après ou plus loin) de l'asymptote, et par conséquent, il faut ajouter ε_i qui décroît quand $y \to \pm\infty$ et la droite aussi tend vers $\pm\infty$, tandis que dans la partie négative de l'axe X le graphe de la tangente précéde l'asymptote, et donc il faut soustraire ε_i qui décroît de la même façon qu'en partie positive.



Vu que le domaine du problème est $x \in [-1...2]$, la fonction atteint son maximum en:

$x_{19} = 37/20 + \in_{19} = 1.85 + \in_{19}$, où $f(x_{19})$ est légèrement plus grande que:

$f(1.85) = 1.85 * \sin(18\pi + \pi/2) + 1.0 = 2.85$.

Supposons que nous voulons construire un AG pour résoudre le problème ci-dessus, c'est-à-dire maximiser la fonction f.Commençons d'abord par décrire les composants principaux de cet AG.

- **Représentation et encodage**

Nous utilisons un vecteur binaire comme un chromosome pour représenter les valeurs réelles de la variable x. La longeur du chromosome dépend de la précision voulue, qui est, dans notre exemple, six positions après la virgule.La longueur du domaine de la variable x est 3; l'exigence de cette précision implique que l'intervalle [-1...2] doit être divisé en, au moins, 3000000 intervalles égaux. Cela veut dire que 22 bits sont nécessaires pour le vecteur binaire (le chromosome):

$$2097152 = 2^{21} < 3000000 \leq 2^{22} = 4194304$$

Etablir la correspondance-the mapping- entre la chaîne binaire $(b_{21}b_{20}...b_0)$ et le nombre réel x qui appartient à l'intervalle [-1...2] est simple, et se passe en 2 phases:

1) Convertir la chaîne binaire$<b_{21}b_{20}...b_0>$ de base 2 en base 10:

$(<b_{21}b_{20}...b_0>)_2 = (\Sigma_i\ b_i * 2^i)_{10} = x'$, pour i = 0,...21.

2) Trouver le réel x correspondant:

$$x = -1.0 + (x' * [3/(2^{22}-1)])$$

Où -1.0 est la limite à gauche du domaine, et 3 est la longueur du domaone.

Par exemple, le chromosome(1000101110110101000111) représente le nombre 0.637197 car

$x' = (1000101110110101000111)_2 = 2288967$ et

$x = -1.0 + 2288967 * (3/4194303) = 0.637197.$

Bien entendu, les chromosomes (0000000000000000000000) et (1111111111111111111111) représentent les limites de domaine, -1.0 et 2.0, respectivement.

$$[(00...00)_2 = 0 = x',\ x = -1.0 + 0 * 3/4194303 = -1.0].$$

- **La population initiale**

Le processus d'initialisation (ou de génération) de la population initiale est très simple: on crée une population de chromosomes, où chaque chromosome est un vecteur de 22 bits. Tous les bits de chaque chromosome sont initialisés (générés) aléatoirement.

- **La fonction d'évaluation**

La fonction d'évaluation *eval* des vecteurs binaires v est équivalent à la fonction f:

$$eval(v) = f(x),$$

Où le chromosome v représente la valeur réelle x (calculée selon la procédure précédente x' puis le réel x).

La fonction d'évaluation joue le rôle de l'environnement- en génétique biologique et en théorie de l'évolution-, en classant les solutions potentielles proportionnellement à leurs fitness. Par exemple, les trois chromosomes:

$v_1 = (1000101110110101000111)$,

$v_2 = (0000001110000000010000)$,

$v_3 = (1110000000111111000101)$,

correspondent aux valeurs $x_1 = 0.637197$, $x_2 = -0.958973$, et $x_3 = 1.627888$ respectivement. Par conséquent, la fonction d'évaluation les classe de la façon suivante:

$eval(v_1) = f(x_1) = x_1 * \sin(10\pi\ x_1) + 1.0 = 1.586345$,

$eval(v_2) = f(x_2) = 0.078878$,

$eval(v_3) = f(x_3) = 2.250650$.

Clairement, le chromosome v_3 est le meilleur des trois, car son évaluation est la plus haute.

- **Les opérateurs génétiques**

Nous utilisons pour la phase de reproduction de l'AG deux opérateurs génétiques classiques: le crossover et la mutation. Comme nous l'avons déjà décrit, la mutation altère un plusiuers gènes (représentés par les positions dans le chromosomes) avec une probabilité égale au taux de mutation. Supposons que le cinquième gène du chromosome v_3 était sélectionné pour la mutation. Vu que ce gène est 0, alors il sera flippé (changé) à 1, ainsi le chromosome v_3 sera après la mutation:

v_3' = (11101000001111111000101).

Ce chromosome représente la valeur x_3' = 1.721638 et $f(x_3')$ = -0.082257. Cela veut dire que cette mutation particulière a eu comme résultat une diminution significative de la valeur du chromosome v_3. D'un autre côté, si le 10éme gène du v_3 était sélectionné pour la mutation alors:

v_3'' = (11100000011111111000101)

qui correspond à la valeur x_3'' = 1.630818 et $f(x_3'')$ = 2.343555, ce qui est une amélioration de la valeur $f(x_3)$ = 2.250650.

Nous allons illustrer l'opérateur du crossover en l'appliquant aux chromosomes v_2 et v_3 . Supposons que le point de crossover était- alétoirement- sélectionné après le cinquième gène:

v_2 = (00000|00000111111000101),

v_3 = (11100|01110000000010000).

les deux descendants résultants sont:

v_2' = (00000|00000111111000101),

v_3' = (11100|01110000000010000).

L'évaluation de ces descendants donne:

$f(v_2')$ = f(-0.998113) = 0.940865,

$f(v_3')$ = f(1.666028) = 2.459245.

Notons que le deuxième descendant v_3'a une meilleure évaluation que ses deux parents(0.078878 et 2.250650).

- **Les paramètres**

Ici on entend par paramètres, les paramètres de l'AG. Pour ce problème particulier, nous avons utilisé les paramètres suivants:

La taille de la population pop_size = 50, la probabilité du crossover p_c = 0.25, la probabilité de mutation p_m = 0.01.

- **Les résultats de l'expérience**

Dans le tableau suivant nous donnons le nombre de générations pour qui nous avons noté une amélioration de la fonction d'évaluation, ensemble avec la valeur de cette fonction. Le meilleur chromosome obtenu après 150 générations était:

v_{max} = (11110011010001000000101), qui correspond à la valeur x_{max} = 1.850773.

Comme c'était prévu- par le calcul théorique-, $x_{max} = 1.85 + \in$, et $f(x_{max})$ est légérement supérieure à 2.85.

Nombre de générations	fonction d'évaluation
1	1.441942
6	2.250003
8	2.250283
9	2.250284
10	2.250363
12	2.328077
39	2.344251
40	2.345087
51	2.738930
99	2.849246
137	2.850217
145	2.850227

Les résultats de 150 générations

- **Implémentation**

Pour implémenter cet AG, onpeut utiliser les procédures et les fonctions qu'on a utilisé au §4.3. pour implémenter la génération- ou l'initialisation- de la population, le crossover et la mutation; en ce qui concerne la fonction d'évaluation, la fonction *decode* (qui correspond à la première phase de l'évaluation) ne change pas et on peut l'utiliser comme elle est, tandis qu'il faut adapter la fonction de fitness ("objfunc") de la façon suivante:

```
function objfunc( x : real) : real;
   { fonction de fitness- f(x) = x * sin(10πx) + 1.0 }
var
    x1 : real;
begin
        x1 := -1.0 + x *(3/4194303);
        objfunc := x1 * sin(10πx1) + 1.0
end;
```

Bien entendu, il faut changer quelques valeurs constantes, ainsi maxstring = 22 (au lieu de 30), maxpop = 50 (au lieu de 100), maxgen = 150,...et opérer quelques autres petites modifications.

2.7. Les Réseaux de Neurones(RN)

Avant de parler de la combinaison, on est obligé de commencer par définir et décrire les RN.

2.7.1. Origine biologique

Les réseaux de neurones sont des groupes de cellules du cerveau interagissant les unes avec les autres appelées **neurones**. Chaque neurone est un processeur d'information extrêmement complexe. Un neurone peut contenir une certaine charge électrique avant de "lancer" un signal aux neurones auxquels il est connecté. Ce signal augmente alors la charge électrique de ces neurones.

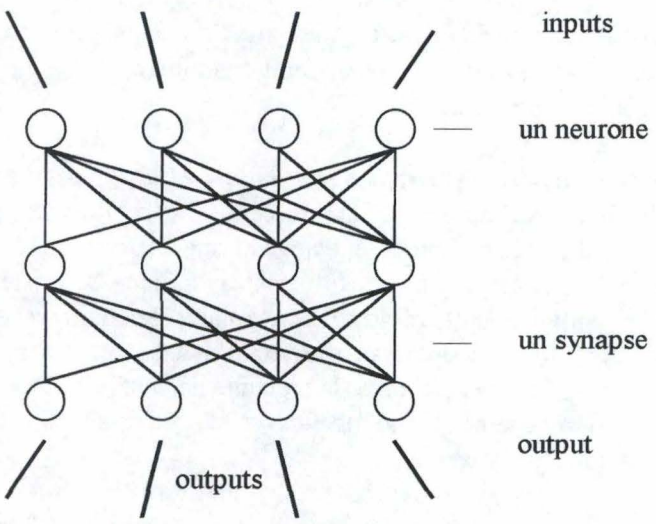
Les neurones sont connectés entre eux par de longues "tentacules" partant du noyau de la cellule. Ces dernières se divisent en plusieurs branches qui se terminent par des petits morceaux appelés **synapses**. Ces synapses peuvent s'accrocher à une partie des autres neurones, établissant un lien électrique et chimique entre les neurones. De cette façon chaque neurone peut être connecté à environ autres neurones.

Certains neurones peuvent également communiquer par d'autres moyens tels qu'en émettant du gaz. Celui-ci peut dès lors affecter d'autres neurones avec lesquels il n' a pas de lien synaptiques. Ainsi, chaque neurone est influencé par beaucoup d'autres, illustrant le fait qu'il n'y a pas de "siège central" ou de "centre d'influence" dans le cerveau.

Les processus complexes de raisonnement du cerveau sont le résultat de nombreuses interactions de ces cellules. Le cerveau humain contient environ 100 milliards de neurones qui ont chacun entre 1.000 et 10.000 connections synaptiques. En comparaison, le ver nématode *Caenorhabditis elegans* a seulement 302 neurones mais peut bouger, réagir à des stimuli et même exécuter des modèles simples de reconnaissance afin de trouver de la nourriture .

2.7.2. La modélisation

Le désir de concevoir une vraie intelligence artificielle existe depuis longtemps. Et qu'y a-t-il de mieux pour créer une infrastructure d'intelligence que d'émuler un schéma qui a fait ses preuves en termes de fonctionnement et d'évolutivité ? Les réseaux de neurones artificiels sont généralement des modèles fortement simplifiés des réseaux de neurones naturels. Ils sont habituellement représentés dans un ordinateur à l'aide d'une grille de neurones et de synapses.



Un exemple de neurones artificiels

Un réseau de neurones artificiel est ordonné en couches, contrairement aux réseaux naturels qui sont dépourvus de toutes structures, afin de simuler le parallélisme des réseaux de neurones naturels. Ainsi, celui-ci est composé de trois types de couches:

⇒ une couche d'entée qui contient les neurones stimulés par l'environnement extérieur ;
⇒ une ou plusieurs couches "cachées" n'ayant pas de lien avec l'extérieur mais rendant l'ensemble du réseau plus complexe et donc plus puissant ;

⇒ une couche de sortie qui contient les neurones qui, une fois activés, engendrent des actions sur l'environnement .

Les réseaux simples ne contiennent d'habitude pas de couche cachée. Une couche peut comprendre autant de neurones que nécessaire et chaque neurone est connecté à tous les neurones de la couche suivante via des synapses. Dans la représentation ci-dessus, il y a une couche d'entrée de 4 neurones, une couche cachée de 4 neurones et également une couche de sortie de 4 neurones. Il y a donc $2*4*4 = 32$ synapses.

Le passage d'une charge électrique est simulée par le fait que les couches sont traitées l'une après l'autre, chaque couche agissant uniquement sur la suivante.

La plupart des modèles ne retirent du fonctionnement réel que les principes suivants :

- les réseaux de neurones sont caractérisés par des interconnexions denses entre des unités de traitement simples agissant en parallèle;
- à chaque connexion est associé un poids qui détermine l'influence réciproque des 2 unités connectées;
- les poids des connexions sont modifiables et c'est cette plasticité qui donne lieu aux facultés d'adaptation et d'apprentissage.

En fait, les dynamiques d'états et de paramètres dépendent fort du modèle de réseau choisi. Dans les modèles les plus populaires, la dynamique des états (appelés pour la circonstance activations des neurones) est donnée par:

$$\mathbf{x_j(t+1) = S(\sum w_{ij} * x_i(t) - \theta_j)}, \text{ où on somme}(\sum) \text{ sur } i=1,...n. \quad (1)$$

où **S(x)** est une fonction sigmoïdale(par exemple, tanh(x)), $\mathbf{x_i(t)}$ l'activation du noeud i à l'instant t, **Wij** le poids de la connexion reliant le noeud i au noeud j, et θ_j un seuil (paramètre de noeud). Chaque neurone est relié à tous les autres (connectivité complète).

Quelquefois (cas du modèle de Hopfield), ce modèle est à dynamique unique et les poids, de même que les seuils, sont fixés *a priori* grâce à un codage approprié du problème à résoudre. On peut alors associer au réseau un fonction d'énergie dont on est certain qu'elle ne peut que décroître; on atteint donc un état stationnaire qui est la réponse au problème fourni par le réseau Hopfield. Le modèle, ou des variantes, sont généralement employés en optimisation (problème du voyageur de commerce, problème de planification, etc.), en reconnaissance de forme (reconnaissance automatique de caractères, de visage, etc.) ou comme mémoire associative. Des modèles plus élaborés (la machine de Boltzmann par exemple) permettent néanmoins une modification des paramètres de noeuds et de poids,suivant des règles d'apprentissage complexes basées sur des calculs statistiques.

D'autres modèles se basent sur des équations d'évolution des états de type "compétitif " (modèle de Kohonen, "Learning Vector Quantization" et "Adaptive Resonance Theory ") et font appel aux propriétés d'auto-organisation qui en découlent et à la non-nécessité d'une entité extérieure fournissant le comportement désiré (apprentissage "non-supervisé").On a d'ailleurs démontré que la plupart des équations de type "compétitif" employées dans ces modèles donnent lieu à des comportements stables, c'est-à-dire tendant vers un point d'équilibre et non pas vers un cycle limite ou un comportement chaotique.Ces modèles sont surtout utilisés en classification automatique. Ils sont soit à dynamique double (de type "à architecture plastique") par l'introduction de nouveaux noeuds si le besoin s'en fait sentir (un nouveau noeud correspond à une nouvelle classe).

Enfin, on utilise intensivement un modèle de réseau particulier, appelé "réseau Perceptron Multi-Couches" (pour le schéma, cf. §2.6.1.2.- "Encodage du réseau"). Il s'agit d'un réseau formé de couches

(groupe de neurones) à connectivité particulière: chaque neurone d'une couche l est relié uniquement aux neurones de la couche précédente(l-1); les équations d'activation sont données encore une fois par l'équation(1) (cf. ci-dessus), où t ne représente plus le temps mais l'indice de la couche. Il ne faut donc pas perdre de vue que la dynamique des états n'est plus qu'un simple calcul de fonction instantané et est donc, en quelque sorte, infiniment rapide. Par exemple, pour un réseau à 2 couches cachées, les valeurs des états des noeuds de la couche de sorite sont données sous forme compacte par :

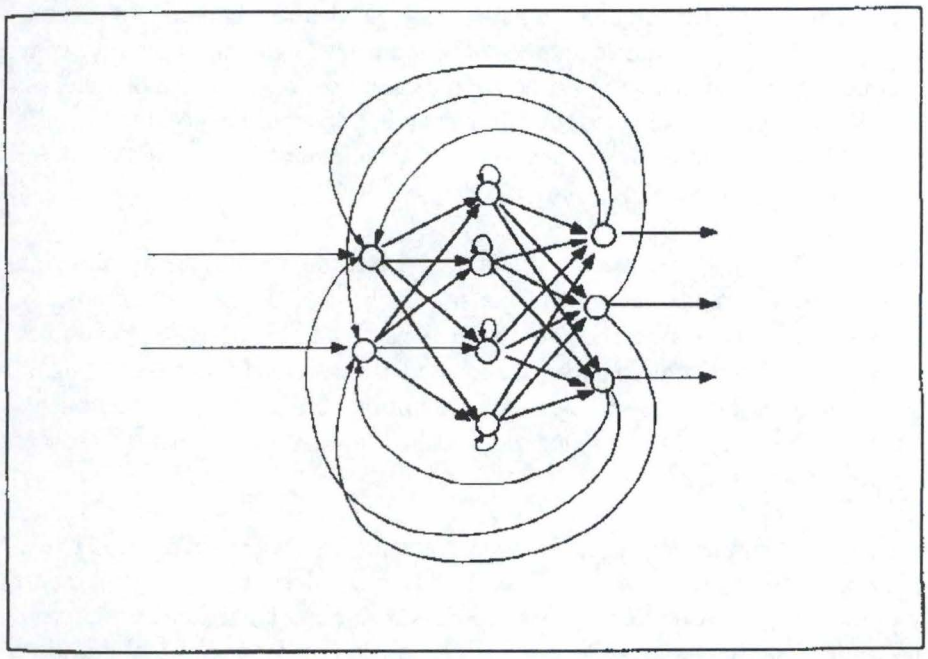
$$\mathbf{o} = \mathbf{W}_3 * \mathbf{S}(\mathbf{W}_2 * \mathbf{S}(\mathbf{W}_1 * \mathbf{e})) \qquad (2)$$

où **o** est le vecteur des états de sortie, **e** le vecteur des signaux d'entrée du réseau, **S** une fonction sigmoïdale, $\mathbf{W}_i$ la matrice de connexion reliant la couche i à la couche i+1 .Dans cet exemple, la dernière couche ne possède pas de fonction sigmoïdale et reste linéaire par rapport aux poids des connexions la reliant à la couche précédente, afin de permettre toutes les excursions possibles du signal de sortie. La dynamique des paramètres est régie par des règles d'apprentissage qui ont pour but de minimiser une fonction d'erreur basée sur un ensemble de vecteurs entrée/sortie désirés, appelés "patrons de référence" ou "patrons d'apprentissage".

Suivant la méthode adoptée pour la minimisation de la fonction d'erreur, on obtient différentes règles d'apprentissage- que nous allons revoir avec un peu plus de détails-, dont la plus célèbre est certainement *l'algorithme de rétroprogation* de l'erreur qui repose sur une minimisation par descente de gradient d'un critère d'erreur de type *moindres-carrés*; il est néanmoins concevable d'utiliser d'autres méthodes de correction des paramètres, par exemple celle du *recuit simulé*(règles d'apprentissage à caractère stochastique).

Enfin, la dynamique de graphe est rarement considérée dans le cas des réseaux multi-couches; cependant, quelques travaux se sont attachés à ce problème en introduisant des principes sélectionnistes dans la formation et l'apprentissage des réseaux neuronaux; la plupart du temps sous forme d'algorithmes génétiques manipulant des graphes peu denses de réseau. D'autres techniques fonctionnent plutôt sur une approche incrémentale, avec élagage éventuel, en augmentant graduellement la taille du réseau suivant ses performances (par exemple, la technique de "Cascade Correlation Learning"). Et comme dans la plupart du temps, l'approche consistant à combiner plusieurs méthodes afin d'additionner leurs points positifs, est une bonne approche, en réseaux neuronaux aussi cette approche peut être utilisée en utilisant le couplage de méthodes faisant varier l'architecture du réseau avec des méthodes de modification des poids des connexions, car ce couplage permet d'obtenir des résultats bien supérieurs aux systèmes dont la seule plasticité réside soit dans les forces de connexion, soit dans l'architecture.

Pour finir, notons deux extensions intéressantes des réseaux perceptrons multi-couches: d'une part les réseaux récurrents (cf. figure ci-dessous) qui permettent des connexions arbitraires entre les noeuds de toutes les couches, même non-adjacentes, et qui réintroduisent une dynamique non-instantanée du genre de l'équation[1] ci-dessus ; ce faisant, les comportements possibles du réseau deviennent beaucoup plus riches: on peut observer des évolutions à points fixes, des cycles limites ou même des attracteurs étranges (chaos). L'étude et les applications de tels réseaux semblent dès lors plus prometteuses, mais aussi beaucoup plus complexes.



Exemple de réseau récurrent

D'autre part, l'emploi de sigmoïdes et de sommes pondérées dans les équations d'activation n'est pas obligatoire; on peut utiliser d'autres fonctions des états de la couche précédente, comme par exemple les fonctions à base radiale; pour résumer (pour plus de détails cf.([REN 95],[WAS 93]), ces réseaux, outre les couches d'entrée et de sortie, comprennent une couche cachée de n unités qui ne réagissent significativement qu'à une partie restreinte de l'espace d'entrée suivant une fonction d'activation de type gaussien. Ces réseaux apprennent vite, et ne souffrent pas du minimum local et autres maladies que la rétroprogation, par exemple, en souffre; mais malheureusement, ils- ou au moins une grande partie parmi eux- exigent que l'ensemble de vecteurs entrée/sortie ("les patrons d'apprentissage") soient impliqués dans leurs opérations, ce qui peut les rendre lents dans l'application finale dont ils font partie.

2.7.3 Fonctionnement

A)Description Générale

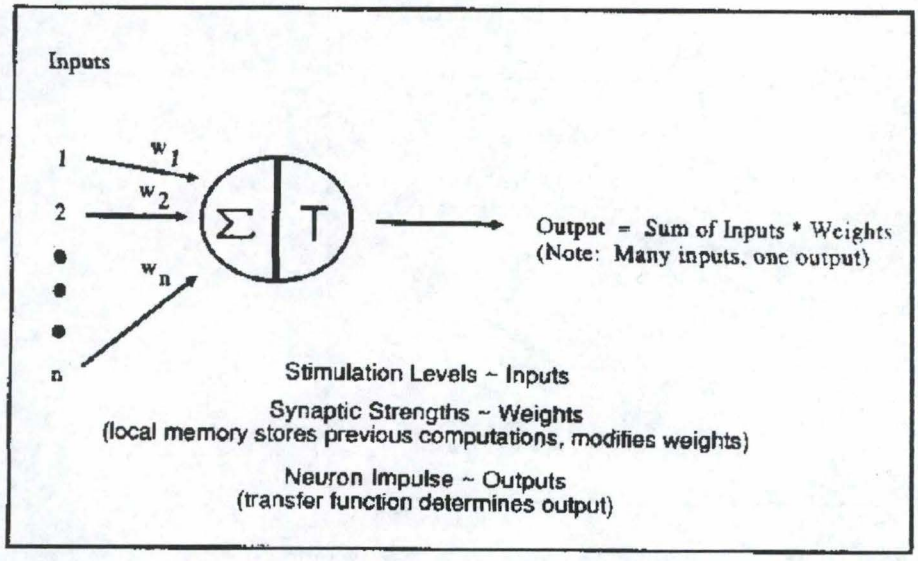
1) **Le réseau**

Le réseau de neurones opère en intervalles de temps discrets. Voici son fonctionnement général :

1. Les neurones de la couche d'entrée sont chargés par les stimuli de l'environnement que le réseau modélise. Ceci est représenté par les flèches en haut du schéma.
2. La charge se propage à travers le réseau de neurones jusqu'à ce qu'elle atteigne la couche de sortie.
3. Si un neurone de la couche de sortie est assez chargé, alors une action de sortie correspondante a lieu. Les flèches en bas du schéma représentent cela.

A chaque synapse est associé un poids qui représente sa "force" c'est-à-dire la puissance à laquelle il propage le signal. Le poids se situe entre -1.0 et $+1.0$. Il est initialisé aléatoirement lors de la création du réseau. Ainsi, la charge passant sur un synapse est d'autant plus affaiblie que le poids de celui-ci est bas. La façon dont la charge se propage à travers le réseau (le "comportement" du réseau) dépend donc entièrement du poids des synapses. Ainsi, on peut attendre d'un robot mobile doté d'un réseau de

neurones que le poids des synapses entre le neurone d'entrée "mur devant" et le neurone de sortie "changer de direction" seront élevés alors que ceux entre "mur devant" et "avancer" seront négatifs
On peut exprimer tout cela sous forme mathématique pour un neurone (voir figure suivante):

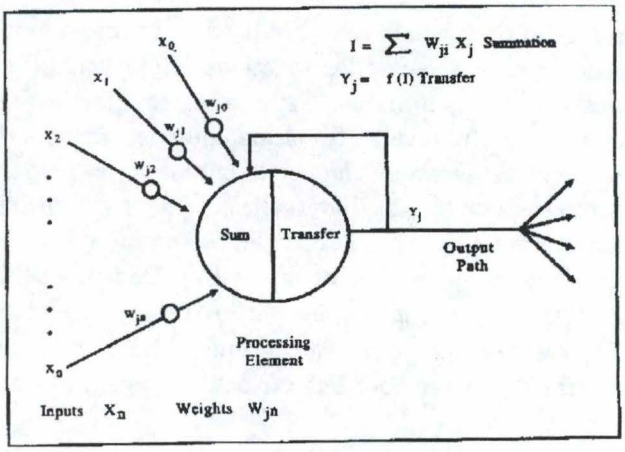


- Entrée : $x_1, x_2,...,x_n$ exprimées sous forme d'un vecteur X
- Poids : $w_1, w_2,...,w_n$ exprimés sous forme d'un vecteur W
- Sortie : NET
- $NET = x_1 .w_1 + x_2.w_2 + ...+x_n.w_n$
- NET = X.W

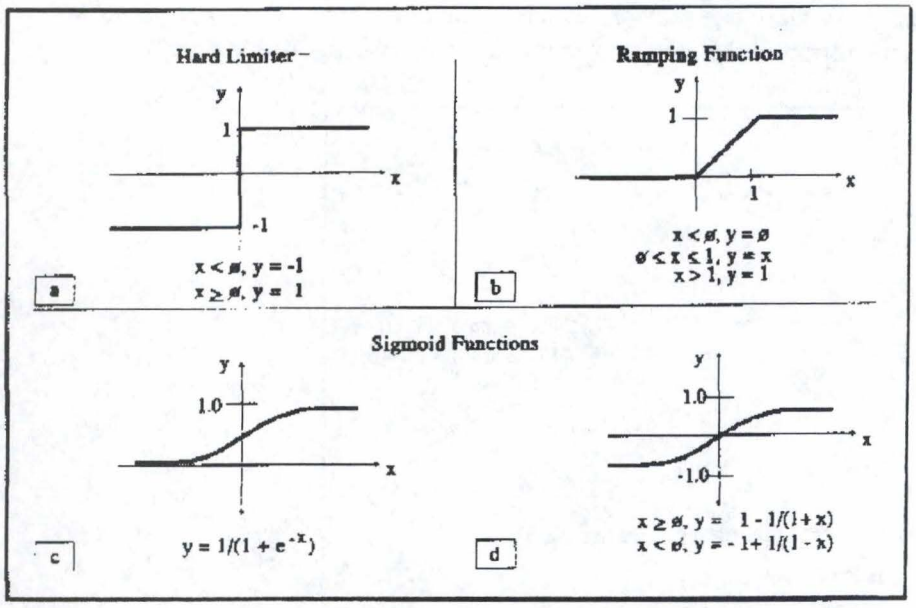
2) **<u>Les fonctions d'activations(ou de transfert)</u>**

- But: Le rôle de la fonction d'activation est de calculer le signal à émettre par un neurone en fonction du résultat contenu dans NET.

OUT = f(NET), où OUT est l'output du neurone.



Ces f[illegible] ..a figure suivante illustre 4 fonctio[illegible]



L'exemple A appartient à une classe de fonctions qu'on appelle "Squatching functions" (fonctions d'applatissement). Avec ces fonctions, quelle que soit la valeur de NET, OUT n'excédera pas une certaine limite. Donc, ce sont des fonctions qui compressent le résultat de NET dans un certain intervalle. Dans notre exemple A, les inputs positifs (0 inclus) produisent un output 1; les inputs négatifs produisent un output –1 (ou 0). On appelle cette fonction "hard limiter" d'output binaire.

L'exemple B illustre une fonction de seuil (ou Ramping function). Les inputs situés entre 0 et 1 sont mappés en output dans le même intervalle (ou parfois ils sont biaisés), et les inputs inférieurs à 0 ou supérieurs à 1 produisent des outputs exactement comme dans l'exemple A. Nous remarquons que les x correspondent aux inputs, et les y correspondent aux outputs de cette fonction. Donc, x ≡ NET, et y ≡ OUT.

La fonction sigmoïde dans l'exemple C est fréquement utilisée. On peut créer plusieurs variantes de cette fonction, en ajustant la pente (l'inclinaison) de la courbe; les dérivées de cette fonction sont faciles à calculer.

L'exemple D représente une fonction similaire à celui de C, mais sans l'utilisation de logarithmes. Ils existent des variations- ou des variantes- de cette fonction, comme les fonctions sin(x), tanh(x) et des autres variantes non-linéaires qui sont eux aussi, parfois, utilisées. Cette fonction d'activation est utilisée comme un limiteur- ou délimiteur- dynamique de l'intervalle. Si on manipule les inputs dans un intervalle approprié, alors la discrimination des inputs est faisable; sans cette limitation, la plupart des données d'input produisent des outputs situés aux extrémités de l'intervalle, ce qui rend difficile la distinction entre un output et un autre. Ceci est vrai dans l'exemple D pour les inputs où $x < -1$ ou $x > +1$. Les valeurs d'output pour $x = 10$ seraient presque les mêmes que pour $x = 100$. De la même façon, les grandes tendances se perdent à cause de la focalisation sur les petites perturbations, ou vice versa. Autrement dit, pour une valeur de NET (ou x) près de 0, un petit changement de NET provoque un grand changement de OUT (ou y); et pour une grande valeur de NET, un petit changement de NET provoque un petit changement de OUT.

Enfin, nous allons décrire très brièvement une fonction d'activation- ou de réponse- fort utilisée, et qui s'appelle: fonction de base radiale. Les réseaux de neurones à fonction de base radiale, que nous appellerons simplement réseaux à base radiale, sont fort utilisés grâce à leur rapidité d'apprentissage, leur généralité et leur simplicité. Outre les couches d'entrée et de sortie, ils comprennent une couche

cachée de n unités qui ne réagissent significativement qu'à une partie restreinte de l'espace d'entrée suivant une fonction d'activation de type gaussien.

Dans le cas simple où il y a seulement une couche cachée et une couche output, le vecteur d'input $X = (x_1, x_2,...,x_m)$ est appliqué à tous les neurones de la couche cachée(m n'est pas égal nécessairement à n). Chaque nuerone de la couche cachée calcule la fonction exponentielle suivante:

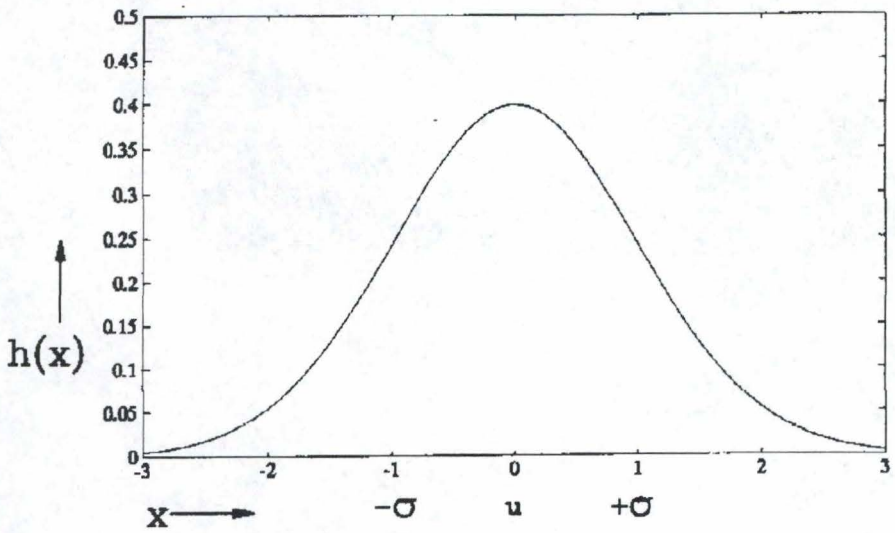
$$h_i = \exp[(-(D_i)^2) / (2\sigma^2)], \text{ où}$$

$(D_i)^2 = (x - u_i)^T (x - u_i)$, x est un vecteur d'input, x et u sont des vecteurs colonnes, u_i est le vecteur de poids du iéme neurone de la couche cahée, et enfin T est le vecteur transposé.

Notons que les poids de chaque neurone de la couche cachée sont assignés les valeurs d'un vecteur input d'apprentissage. L'output du neurone est:

$$y = \Sigma_i h_i w_i \qquad \text{où } w_i \text{ est un poids de la couche de sortie.}$$

Pour mieux comprendre l'opération du réseau, supposons que le neurone possède un seul input x (voir la figure suivante).



Dans ce cas , pour ce neurone la fonction est:

$$h = \exp[(x-u)^2 / 2\sigma^2]$$

Et l'output h(x) varie avec x. Quand x = u, la fonction est 1.0. Donc u détermine la valeur de x qui produit la valeur maximum d'output du neurone; la réponse aux autres valeurs de x chute rapidement quand x dévie de u, et elle devient négligeable quand x s'éloigne de trop de u. Pour cela, l'output a une réponse significative à l'input x, seulement quand x appartient à un intervalle appelé le champ réceptif du neurone; la taille de cet intervalle est déterminée par la valeur de σ. Par analogie à la distribution normale en statistiques qui la même forme que celle de h(x) (courbe en cloche de Gauss), u peut être appelé la moyenne et σ l'écart type.

Bien entendu, on peut généraliser à deux inputs x_1 , x_2 (voir la Figure ci-après), et h sera une fonction de base radiale à 2 dimensions(une cloche, et non une courbe en cloche, de Gauss) telle que:

$h = h(x_1 ,x_2)$, et bien sûr on peut ainsi continuer la généralisation pour plusieurs inputs (ou dimensions).

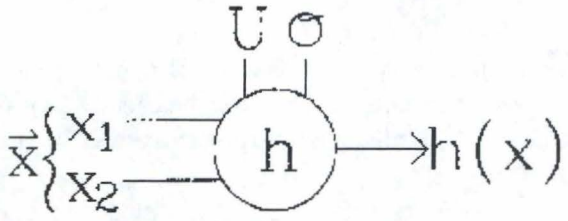
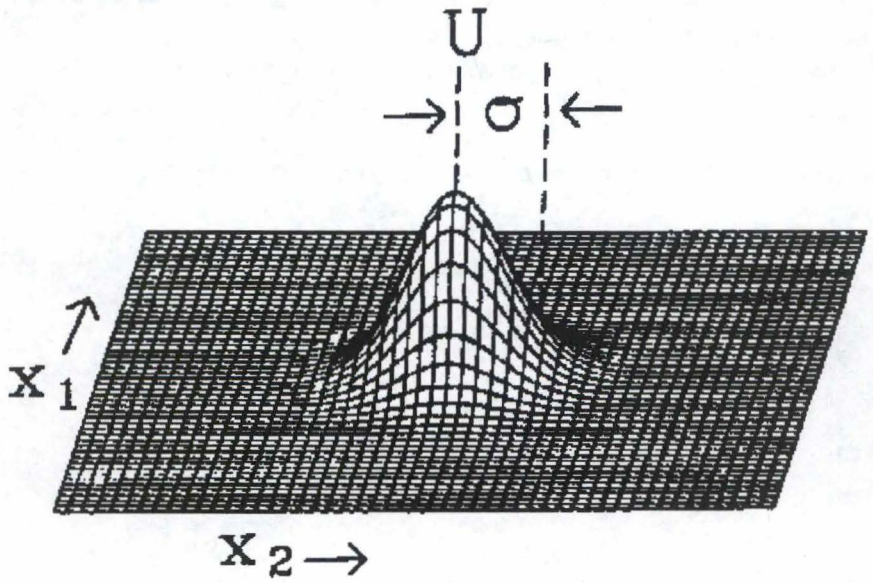


Figure 8-3(a). Simplified single neuron.



B) Les méthodes de propagation

1) Propagation vers l'avant

La propagation de la charge à travers le réseau de neurones est implémentée à l'aide de l'algorithme de propagation vers l'avant. Celui-ci fonctionne de la manière suivante :

1. Affecter aux neurones de la couche d'entée les valeurs booléennes renvoyées par les fonctions modélisant les stimuli de l'environnement extérieur. Par exemple, si la fonction "mur devant" renvoie vrai, alors le neurone correspondant de la couche d'entrée émettra une charge égale à 1 ; si elle renvoie faux, la charge vaudra 0.

2. Propager la charge de chaque neurone aux neurones de la couche suivante :
   a. La charge envoyé par un neurone d'une couche est multipliée par les poids des synapses liant ce neurone aux neurones de la couche suivante
   b. Chaque neurone de la couche suivante reçoit donc autant de valeurs qu'il existe de neurones dans la couche précédente. Ces valeurs sont donc additionnées.
   c. Afin de maintenir le résultat de l'addition entre 0.0 et 1.0, on lui applique une fonction d'activation, le plus souvent la fonction ***sigmoïde*** :

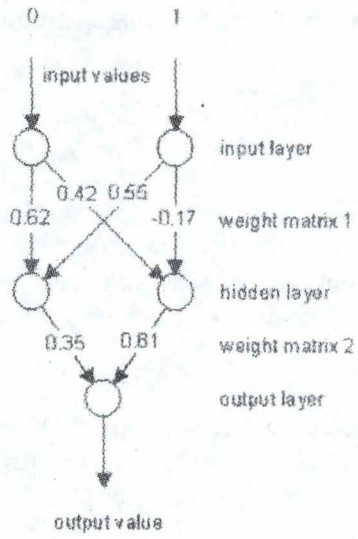
$$f(x) = \frac{1}{1 + e^{-x}}$$

Cette valeur constitue alors la nouvelle charge du neurone qu'il passera à son tour à chaque neurone de la couche suivante par le même procédé.

3.Répéter l'étape 2 jusqu'à la couche de sortie.

4.La valeur de chaque neurone de sortie est comparée à un seuil de tolérance prédéterminé. Si cette valeur est plus grande que le seuil, l'action correspondante au neurone est effectuée (la fonction qui agit sur l'environnement est appelée). Par exemple, avec un seuil de 0.51, si le neurone de sortie "avancer" a une valeur de 0.84, alors la fonction "avancer" est effectivement appelée ; si la valeur vaut 0.2, rien ne se passe.

Exemple :



Soit le réseau illustré ci-dessus : une couche d'entrée de 2 neurones, une couche cachée de 2 neurones et une couche de sortie de 1 neurone. Les poids des synapses ont été initialisés aléatoirement. Examinons la propagation vers l'avant si les valeurs d'entrée sont 0 et 1 :

Input du neurone caché n° 1 :	0 * 0.62 + 1 * 0.55 = 0.55
Output du neurone caché n° 1 :	1 / (1 + exp(-0.55)) = 0.634135591
Input du neurone caché n° 2 :	0 * 0.42 + 1 * (-0.17) = 0.17
Output du neurone caché n° 2 :	1 / (1 + exp(+0.17)) = 0.4576602059
Input du neurone de sortie :	0.634 * 0.35 + 0.457 * 0.81) = 0.592605124
Output du neurone de sortie :	1 / (1 + exp(-0.592605124)) = **0.643962658**

Si **0.643962658** est plus grande que le seuil de tolérance, alors la fonction correspondante au neurone de sortie est appelée.

2) **Propagation vers l'arrière(en anglais BackPropagation ou BP)**

- **Définition**

⇒ C'est un modèle applicable à une grande classe de problèmes. Il est certainement l'algorithme d'apprentissage supervisé le plus utilisé.
⇒ Typiquement, la BP emploie 3 couches d'unités de traitement ou plus.
⇒ Ils ne doivent pas être totalement connectés mais en général, c'est souvent le cas. La figure suivante illustre la topologie de la BP

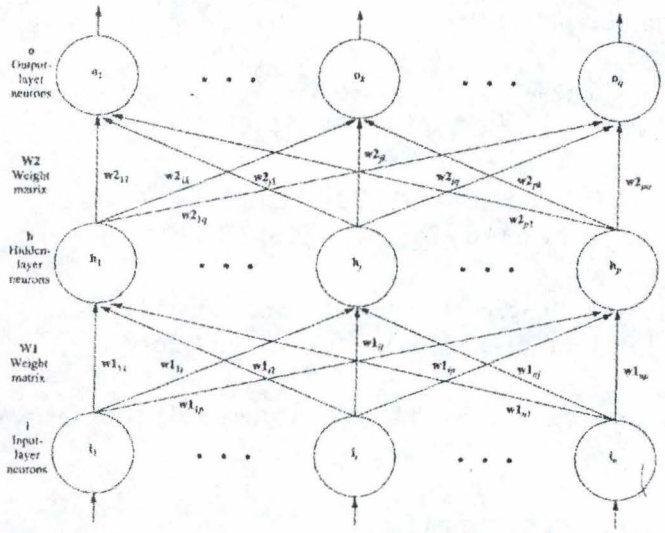
Dans ce modèle, il y a 3 couches de neurones: input, cachée et output; et il y 2 "couches" de poids de synapses. Il y a aussi le taux d'apprentissage α, utilisé dans les calculs ci-dessous. Ce taux indique combien on doit changer les poids pendant chaque étape. Typiquement, c'est un nombre entre 0 et 1. Il y a aussi le moment Θ, qui indique combien le changement précédent des poids influence- ou doit influencer- le changement courant des poids. Enfin, il y a un terme qui indique le domaine de tolérance à l'intérieur duquel on peut accepter un output comme un "bon".
⇒ La BP est entraînée de façon supervisée(cf. paragraphe suivant c). C'est-à-dire avec des paires de patrons(un en entrée et un en sortie). Après chaque présentation d'une paire de patrons, les poids sont ajustés pour diminuer la différence entre ce que le réseau fournit en sortie et l'objectif désiré.

- **Principe de la BP (2 étapes):**

1) **Passage en avant**:
⇒ Il commence avec la présentation d'un patron d'entrée dans la couche d'entrée du réseau qui se propage dans les couches intermédiaires du réseau.
⇒ Ensuite, à chaque niveau successif, les unités de traitement additionnent les valeurs qu'elles reçoivent en entrée et applique une fonction sigmoïde pour calculer ses données en sortie.

2) **Passage en arrière**:
⇒ Il commence avec la comparaison du patron reçu en sortie avec le résultat désiré
⇒ Ensuite, les valeurs des erreurs sont calculées pour les différentes couches en commençant par la couche de sortie et ce calcul permet de changer les poids .

- **La procédure de calcul**



⇒ **Le passage en avant**

1) Calculer les activations des neurones de la couche cachée:

h = F(iW1), où h est le vecteur des neurons de la couche cachée, i le vecteur de la couche input, et W1 est la matrice des poids entre les couches input et cachée.

2) Calculer les activations des neurones de la couche output:

o = F(hW2), où o représente la couche output, h la couche cachée (hidden layer), W2 la matrice des synapses qui connectent les couches cachées et d'output, et F( ) est une fonction d'activation sigmoïde. Nous allons utiliser la fonction logistique:

$$F(x) = 1 / (1 + e^{-x})$$

⇒ **Le passage en arrière**

3) Calculer l'erreur de la couche output (la différence entre l'objectif désiré et l'objectif observé- qui est sortie-).

$$d = o(1-o)(o-t)$$

Où d est le vecteur d'erreurs pour chaque neurone d'output, o est le vecteur de la couche output, et t est l'objectif (correct) d'activation de la couche output.

4) Calculer l'erreur de la couche cachée:

$$e = h(1-h)(W2)d$$

Où e est le vecteur d'erreurs de chaque neurone de la couche cachée.

5) Ajuster les poids de la seconde couche de synapses:

$$W2 = W2 + \Delta W2$$

Où ΔW2 est la matrice représentant le changement dans la matrice W2 . Il est calculé de la façon suivante:

$$\Delta W2_t = \alpha hd + \Theta \Delta W2_{t-1}$$

Répéter les étapes 1 à 6 pour toutes les paires de patrons jusqu'à que l'erreur de la couche output(le vecteur d) soit dans le domaine de tolérance spécifié de chaque patron et de chaque neurone.

- **Le rappel (l'évocation)**

Présenter le patron d'input à la couche d'input du réseau BP (décrit ci-dessus):

1) Calculer l'activation de la couche cachée

$$h = F(W1i)$$

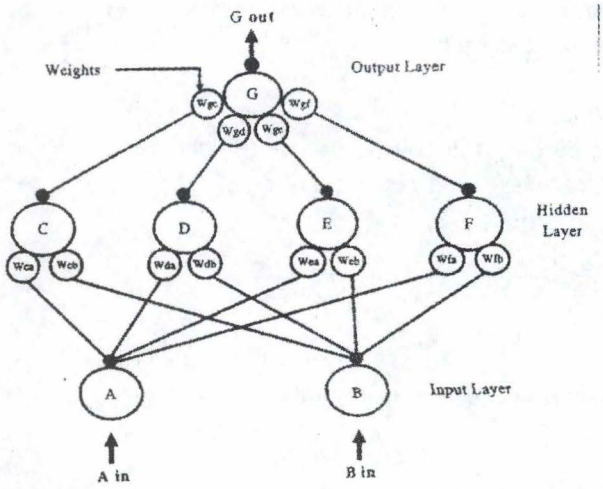
2) Calculer la couche output:

$$o = F(W2h)$$

Le vecteur o est le patron rappelé (évoqué).

- **Exemple**

Dans cet exemple- simple-, nous allons illustrer la méthode BP, et montrer les mécanismes utilisés pour l'ajustement des poids, et l'apprentissage basé sur l'ensemble d'entraînement, afin de produire l'output désiré (ou atteindre l'objectif désiré). La Figure suivante illustre le réseau utilisé dans notre exemple.



Nous avons disposé le réseau de cette façon afin de mettre en évidence les poids associés aux unités de traitement (Processing Element ou PE). Cette façon de représenter les pods n'est pas standard, mais nous l'avons utilisé pour simplifier l'exemple. D'ailleurs, c'est pour la même raison que nous avons omis la fonction d'activation- de transfert- et sa première dérivée; et en plus pour amplifier le processus du changement des poids.

Le réseau contient 7 PE, arrangés en trois couches. La couche d'input contient 2 PE nommés A et B. La couche cachée contient 4 PE: C, D, E et F. La couche output a 1 PE: G. Nous allons décrire les données. Une façon acceptable d'assigner des données aux éléments du réseau est de les choisir aléatoirement. On appelle cette procédure "randomizing" les poids. Nous avons besoin des définitions suivantes:

- Ain désigne une valeur d'input pour le PE "A".
- Gout désigne une valeur d'output pour le PE "G".
- Wca désigne une valeur d'un poids pour le PE "C", venant du PE "A". Autrement dit, c'est le poids du synapse ac
- Eg désigne la valeur d'erreur pour le PE "G".

Nous commençons les calculs par citer la liste de données:

Ain = 0; Bin = 1;

L'output désiré Gout = 1;

Puis, nous allons générer aléatoirement les poids en commençant par la couche output, et en reculant vers la couche input, ce qui donnera:

Wgc = 1.2	Wgd = 1.6	Wge = 4.3	Wgf = 3.2
Wca = 1.1	Wcb = -1.4	Wda = 3.6	Wdb = -4.1
Wea = 2.1	Web = 2.5	Wfa = 0.9	Wfb = -1.0

Afin de pouvoir calculer les "nouvelles" valeurs des poids(après chaque cycle), nous devons connaître l'output de chaque PE, et aussi l'erreur de ce PE. La formule pour calculer les outputs des PE est:
Equation1: (l'actuel) Gout = [Cout*Wgc]+[Dout*Wgd]+[Eout*Wge]+[Fout*Wgf]

Nous devons calculer tous les outputs de tous les PE connectés à G avant de pouvoir calculer Gout.
Equation2: Cout = [Ain*Wca]+[Bin*Wcb] = -1.4
Equation3: Dout = [Ain*Wda]+[Bin*Wdb] = -4.1
Equation4: Eout = [Ain*Wea]+[Bin*Web] = 2.5
Equation5: Fout = [Ain*Wfa]+[Bin*Wfb] = -1.0

Nous pouvons maintenant calculer Gout de l'Equation1, ce qui nous donne Gout = -0.69.
Avant que nous pouvons calculer les 8 poids restants associés avec la couche cachée, nous devons calculer l'erreur pour chaque PE. L'erreur de l'output du réseu est égal à la différence entre l'output désiré et l'output actuel- observé-. Donc l'erreur pour le PE "G" est:
Equation6: Eg = OUTPUT DESIRE - output Actuel = 1.69
Equation7: Ec = Eg*Wgc = 2.03
Equation8: Ed = Eg*Wgd = 2.70
Equation9: Ee = Eg*Wge = 7.27
Equation10: Ef = Eg*Wgf = 5.41

Maitenant, nous pouvons calculer les "nouvelles" valeurs de chacun des 12 poids. Léquation générale pour calculer les nouvelles valeurs des poids est:

NOUVEAU POIDS = ERREUR*OUTPUT + POIDS ACTUEL

Equation11: New Wgc = [Eg*Gout] + CURRENT Wgc = 0.03
Equation12: New Wgd = [Eg*Gout] + CURRENT Wgd = 0.43
Equation13: New Wge = [Eg*Gout] + CURRENT Wge = 3.13
Equation14: New Wgf = [Eg*Gout] + CURRENT Wgf = 2.03
Equation15: New Wca = [Ec*Cout] + CURRENT Wca = -1.74
Equation16: New Wcb = [Ec*Cout] + CURRENT Wcb = -4.24
Equation17: New Wda = [Ed*Dout] + CURRENT Wda = -7.47
Equation18: New Wdb = [Ed*Dout] + CURRENT Wdb = -15.17
Equation19: New Wea = [Ee*Eout] + CURRENT Wea = 20.28
Equation20: New Web = [Ee*Eout] + CURRENT Web = 20.68
Equation21: New Wfa = [Ef*Fout] + CURRENT Wfa = -4.51
Equation22: New Wfb = [Ef*Fout] + CURRENT Wfb = -6.41

Dans le tableau suivant, on peut comparer les poids qu'on vient de calculer avec les poids originaux générés aléatoirement:

PE	Poids	Valeur originale	Nouvelle valeur
C	Wca	1.1	-1.74
C	Wcb	-1.4	-4.24
D	Wda	3.6	-7.47
D	Wdb	-4.1	-15.17
E	Wea	2.1	20.28
E	Web	2.5	20.68
F	Wfa	0.9	-4.51
F	Wfb	-1.0	-6.41
G	Wgc	1.2	0.03
G	Wgd	1.6	0.43
G	Wge	4.3	3.13
G	Wgf	3.2	2.03

Et nous devons continuer à répéter ces cycles jusqu'à que Gout soit égal à 1 (comme dans notre cas l'actuel ou dernier Gout était égal à -0.69), ou soit égal, par exemple, à 1±0.05 dans d'autres cas.

C) L'apprentissage

1) Le principe

L'apprentissage d'un réseau neuronal est le processus d'entraînement qui possède un ensemble de mécanismes permettant l'amélioration des performances futures du réseau sur la base d'une connaissance acquise par ce réseau au fur et à mesure de ses expériences passées.

L'apprentissage requiert essentiellement 3 caractéristiques de la part du réseau et de son environnement:

- Une structure permettant de mettre en oeuvre une association ou une fonction (au sens mathématique du terme, en anglais "mapping") correspondant à une loi de commande, à un comportement ou un réflexe, à une reconnaissance, etc;
- Une capacité d'ajustement des poids (w) des connexions entre unités;
- Une mesure de performance et sa rétroaction;

Si nous adoptons le critère comportemental pour décrire le système d'apprentissage, alors nous pouvons décrire mathématiquement- et dans notre cas brièvement- ce système et ses 3 caractéristiques citées ci-dessus. Nous disons que le système a appris la paire stimulus-réponse (x_i, y_i) si il répond par y_i quand x_i le stimule. x_i peut représenter une distribution spectrale (qui a des rapports avec les séries et la transformation de Fourier), et y_i désigne une classe de patrons. La paire input-output (x_i, y_i) représente un exemple de la fonction f: $R^n \rightarrow R^p$. La fonction f correspond (maps) des n-vecteurs x aux p-vecteurs y.

On dit que le système a appris la fonction f, si ce système répond avec y, où y=f(x), quand on lui présente x, et cela quelque soit x.Le système a partiellement- ou approximativement- appris la fonction f, si le système répond par y', qui est "proche" de y=f(x), quand on lui a présenté x', qui est proche de x. Le système correspond des outputs similaires aux inputs similaires, et ainsi il évalue- il estime- une fonction continue. En fait, un bon apprentissage il doit être monotone et consistant.

L'apprentissage implique un changement. Un système apprend, s'adapte ou "s'auto-organise" quand le changement d'échantillons de données changent les paramètres du système. En NN, l'apprentissage peut être n'importe quel changement dans n'importe quelle synapse.

L'apprentissage implique aussi la quantification . D'habitude, le système apprend une petite part de tous les patrons de l'environnement échantillonné. Le nombre d'échantillons peut bien être infini. D'ailleurs, l'utilisation de l'indexation discrète dans la notation(x_i, y_i) reflète la grande différence entre le nombre de patrons possibles et le nombre de patrons appris. En général, il y a un continuum d'un grand nombre de fonctions d'échantillonage, ou un vecteur-aléatoire de réalisations,(x,y).

Vu que la mémoire est limitée pour n'importe quel système, alors l'apprentissage utilise le remplacement des patrons anciens par des nouveaux patrons, et la construction des "représentations internes" ou des prototypes des patrons échantillonnés. Les modèles des NN représentent les échantillons de prototypes par des vecteurs de nombres réels.

2) Les modes d'apprentissage

En ce qui concerne la troisième caractéristique de l'apprentissage, supposons que nous disposons d'une fonction J(w), traduisant les objectifs de l'apprentissage. Fondamentalement, les modifications des poids de connexions peuvent se faire suivant deux mécanismes de rétroaction:

- **par gradient** : le mécanisme nécessite la connaissance de la quantité $\partial J(w) / \partial w$,éventuellement sous forme approchée, qui fournit une information de direction et d'amplitude du changement à effectuer sur les poids w pour améliorer J (apprentissage supervisé);
- **sans gradient** (plus général): lorsque l'information du gradient n'est pas disponible (quand les objectifs sont vagues ou "mal définis", etc), il faut recourir à des mécanismes d'apprentissage par renforcemen, qui fonctionnent sur la base d'une exploration judicieusement organisée de toutes les

sorties-outputs- possibles du réseau et en déduisent les variations des poids w (apprentissage non-supervisé).

Examinons un peu plus en détails le mécanisme d'apprentissage de type gradient. Supposons donc disposer d'un réseau réalisant la transformation(paramétrisée): $y = N(u;w)$ où y est la sortie du réseau ($y \in R^p$), u en est l'entrée ($u \in R^m$) et w, les poids ajustables des connexions du réseau ($w \in R^n$). On dispose en outre d'une fonction d'objectif- de performance- J. La modification des poids s'effectue alors suivant une loi du type:

$$\Delta w = -H*(\partial J/\partial w)$$

Où la matrice H peut prendre différentes formes suivant la complexité de l'algorithme d'apprentissage (simple matrice identité I multipliée par une constante η, ou matrice Hessiennne inverse pour des algorithmes plus évolués).

L'objectif est généralement donné sous la forme $J = J(y)$. Par conséquent le gradient $\partial J/\partial w$ s'obtient symboliquement par:

$$\partial J/\partial w = (\partial J/\partial y)*(\partial y/\partial u)*(\partial u/\partial w)$$

Il est donc nécessaire de connaître, au moins approximativement, la quantité $\partial y/\partial u$ appelée "Jacobien du processus", ce qui peut s'avérer problématique lorsque rien n'est connu du processus. D'autre part, la quantité $\partial u/\partial w$, dépend de l'architecture du réseau neuronal utilisé ; si l'architecture est celle d'un perceptron à couches multiples- on va le voir ci-après comme exemple de l'apprentissage supervisé-, le calcul de cette dérivée se fait traditionnellement au moyen de BP- cf. §B.2-.

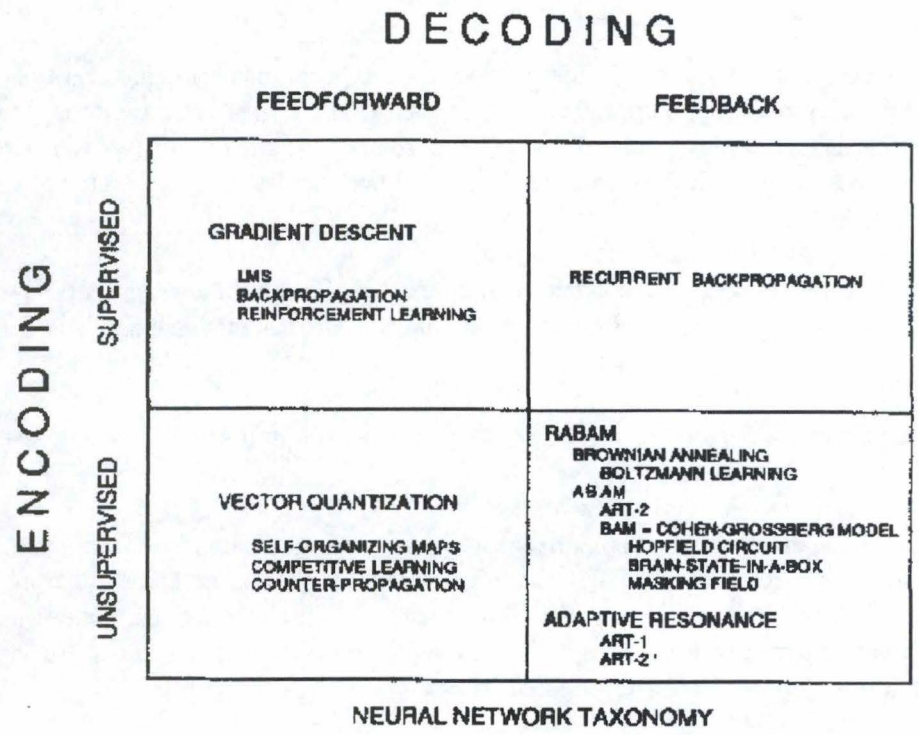
On peut résumer les caractéristiques des deux modes d'apprentissage, par les deux listes non-exhaustives et le tableau suivants:

- **Apprentissage supervisé**

⇒ l'apprentissage supervisé nécessite la mise en correspondance de chaque vecteur d'input avec un vecteur cible représentant l'output désiré. Ensemble, cet input et cet output forment une paire d'entraînement.
⇒ un "professeur" externe est donc nécessaire.
⇒ il faut entraîner le réseau avant qu'il devient opérationel.
⇒ la phase d'apprentissage peut prendre beaucoup de temps. Elle est achevée quand le réseau produit- fournit- l'output désiré en réponse à l'input correspondant

- **Apprentissage non-supervisé**

⇒ modèle d'apprentissage beaucoup plus plausible biologiquement
⇒ pas de vecteur cible d'output et donc pas de comparaisons(et donc pas de professeur externe); et pour cela on appelle cet apprentissage "apprentissage auto-supervisé". A la place, il y a un monitoring interne de la performance.
⇒ l'ensemble d'entraînement consiste simplement en un ensemble de vecteurs d'inputs.
⇒ le processus d'entaînement consiste donc à extraire les propriétés statistiques de l'ensemble d'entraînement et regrouper les vecteurs ressemblants en classes.
⇒il est donc impossible de déterminer avant l'entraînement quel output sera produit pour un input déterminé en entrée ou de dire quelles classes seront formées



2.1) *Apprentissage supervisé*

L'apprentissage d'un réseau est dit supervisé si l'output désiré est connu préalablement.
Exemple : association de patterns

Supposons qu'un réseau doive apprendre à associer les paires suivantes de patterns. Les patterns d'entrée sont des nombres décimaux, chacun représenté par une séquence de bits.

Les patterns cibles sont donnés sous forme de valeurs binaires des nombres décimaux :

	Patterns d'entrée	**Patterns cibles**
1	0001	001
2	0010	010
3	0100	011
4	1000	100

Vu le format des patterns d'entrée et des patterns cibles, on peut s'attendre à un réseau pourvu d'une couche d'entrée de ' neurones (un par bit du pattern d'entrée) et d'une couche de sortie de 3 neurones (idem).

Au cours de l'apprentissage, un des patterns d'entée est donné à la couche d'entrée du réseau. Ce pattern est propagé à travers le réseau (indépendamment de sa structure) jusqu'à la couche de sortie. Cette dernière génère un pattern de sortie qui est alors comparé au pattern cible. En fonction de la différence entre les deux , une valeur d'erreur est calculée .Cette erreur est utilisée pour modifier le poids des synapses et constitue l'effort d'apprentissage du réseau. Plus grande est la valeur d'erreur , plus les poids des synapses seront modifiés.

L'algorithme principal d'apprentissage supervisé est l'algorithme de propagation vers l'arrière qu'on a étudié en B).Cet algorithme implémente les méthodes données dans l'exemple ci-dessus Il garantit qu'à chaque exécution, la valeur d'erreur du réseau diminue. Nous allons étudier un autre exemple de l'apprentissage supervisé qu'est le perceptron.

◊ **Le perceptron**

- le perceptron est une architecture neuronale qui apprend à classifier des patrons grâce à un entraînement supervisé .
- les patrons qu'il classifie de même que les outputs (les catégories de classification) sont souvent des vecteurs de valeur binaires .
- il est constitué de deux couches de neurones avec une couche de poids adaptables entre ces deux couches .

Il peut éventuellement comprendre plus de deux couches de neurones mais une seule couche de poids est capable de s'adapter .

Activation d'un neurone

$S_j = \Sigma^n_{i=0}\ a_i\ w_{ji}$
avec a_i la valeur d'input de l'unité i
w_{ji} le poids associé à la connection entre l'unité i et l'unité j

Fonction d'activation

- la fonction d'activation est une fonction de seuil
  if $S_j > 0$ then $x_j = 1$
  if $S_j \leq 0$ then $x_j = 0$
  avec x_j la valeur d'output de l'unité j

Ajustement des poids

- l'ajustement des poids des connections du perceptron peut se faire par une multitude d'algorithmes d'apprentissage (apprentissage supervisé).

Par exemple :
new w_{ji} = old w_{ji} + C $(t_j - x_j)a_i$
avec t_j la valeur désirée de l'unité d'output j
x_j la valeur effective de l'unité d'output j
$(t_j - x_j) = 1$ si $t_j = 1$ et $x_j = 0$
0 si $t_j = x_j$
-1 si $t_j = 0$ et $x_j = 1$
$a_i = 1$ ou 0 , la valeur de l'unité d'input i
- C le "taux d'apprentissage "
-un poids n'est ajusté que si son unité d'input est active(c'est-à-dire $a_i = 1$)
-le taux d'apprentissage est une petite constante, souvent ≤ 1 et détermine l'importance de la correction effectuée lors d'une seule itération

Performance

- la performance du réseau est mesurée par le niveau d'erreur , lui même calculé par la formule :

$$[\Sigma_p \Sigma_j ((t_{jp} - x_{jp})^2 / (n_p.n_0)]^{1/2}$$

Avec n_p le nombre de patrons dans l'ensemble d'entraînement, n_o le nombre d'unités dans la couche d'output

Limitations

- le perceptron ne permet qu'à une couche de connections d'ajuster ses poids
- une unité d'output ne peut classifier que des patrons linéairement séparables .
  Exemple : la fonction XOR

  REM 1 : difficile de savoir si une fonction est linéairement séparable quand celle-ci a beaucoup de variables $\Rightarrow$ le perceptron est limité à des simples problèmes

  REM 2 : inutile d'établir une deuxième couche de neurones si les fonctions d'activation entre les couches sont linéaires $\Rightarrow$ importance des fonctions d'activation non-linéaires pour étendre les capacités des réseaux au-delà de celles d'un réseau mono-couche .

2.2) ***Apprentissage non-supervisé***

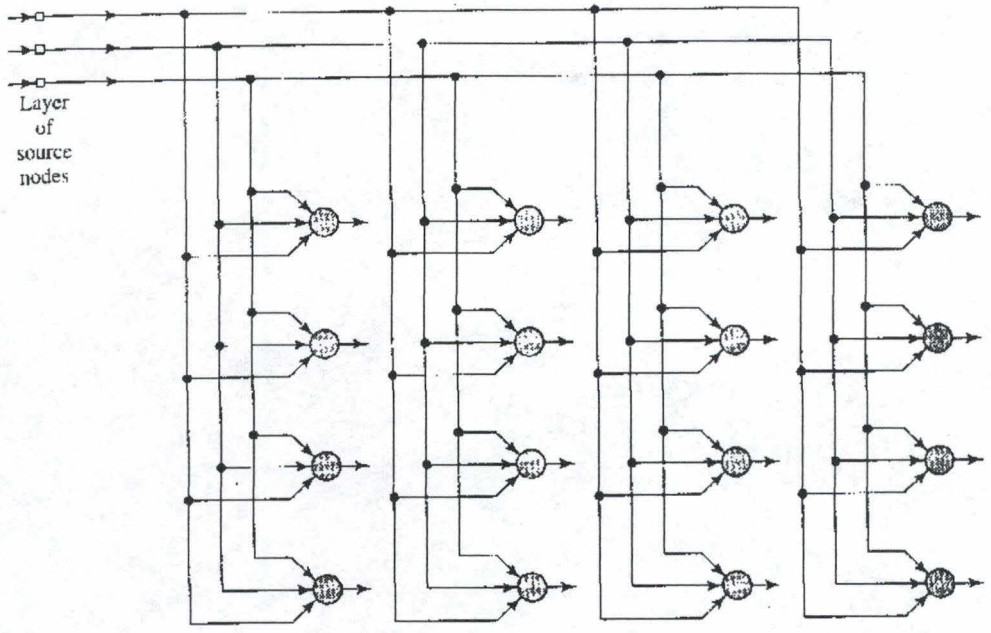
Dans ce type d'apprentissage, le réseau cherche les régularités ou les tendances dans les signaux- ou les données- de l'input, et puis il fait des adaptations selon sa fonction ou son objectif. Un algorithme d'apprentissage non-supervisé (ANS) peut envisager une coopération entre des groupes ou des rassemblements d'unités de traitement (processing elements, PE). Dans ce schéma, les groupes vont travailler ensemble et stimuler les uns les autres. Si un input externe- et pas entre les unités- fait activer un noeud- ou une unité- dans un groupe, alors l'activité du groupe entier sera augmenté. Et de la même façon, si l'input externe de plusieurs noeuds du groupe a baissé d'intensité, cela peut avoir un effet inhibiteur sur le groupe tout entier, et ainsi ces groupes peuvent alors compresser les données avant de les transmettre à la couche suivante.

La compétition entre les PE peut aussi former la base de l'ANS. L'apprentissage des groupes en compétition peut amplifier les réponses des groupes bien spécifiques aux stimuli spécifiques, et associer ces groupes les uns avec les autres d'un côté, avec une réponse spécifique appropriée de l'autre côté.En fait, les systèmes d'ANS rassemblent d'une façon adaptative les patrons en groupes ou classes de décision. Les systèmes d'apprentissage compétitifs font évoluer les neurones "gagants" de la compétition neuronale pour l'activation. Cette compétition est produite- induite- par les patrons d'input échantillonnés aléatoirement. Puis, les vecteurs des liens synaptiques vont estimer les centroïdes des classes de patrons. Les centroïdes dépendent clairement de la densité, inconnue, de la probabilité des patrons p(x). D'autres systèmes neuronaux non-supervisés dévellopent des bassins attracteurs dans l'espace d'états des patrons. Les bassins attracteurs correspondent aux différentes classes de patrons. La dynamique rétroactive attribue aux bassins leurs tailles, positions et nombres.

Les algorithmes de l'ANS ont souvent une complexité de calcul et une exactitude inférieures à celles des algorithmes d'apprentissage supervisé, mais ils apprennent rapidement, souvent avec un seul passage- un cycle-, sur des données avec bruits. Cela rend l'ANS très pratique dans plusieurs environnements à temps réel- des applications à temps réel- et à haute vitesse, où éventuellement on n'a pas assez de temps, d'information ou de précision de calcul pour utiliser les techniques supervisées

Nous allons, à présent, étudier, très brièvement, un des systèmes d'auto-organisation- ANS-, qui est une sous-classe de réseaux neuronaux connue sous le nom: les fonctions de correspondance auto-organisées (self-organizing maps).Ces réseaux sont basés sur l'apprentissage compétitif, donc ANS.

Dans une telle fonction (self-organizing map), les neurones sont placés dans les noeuds d'un treillis, habituellement, d'une dimension- une chaîne-, ou de deux dimensions (voir figure suivante), malgré que plusieurs dimensions est possible, mais ce n'est pas très fréquent. Les neurones sont sélectivement tunés- ajustés- aux différents patrons d'input (stimuli), ou aux classes de patrons d'input, durant le processus d'apprentissage compétitif.



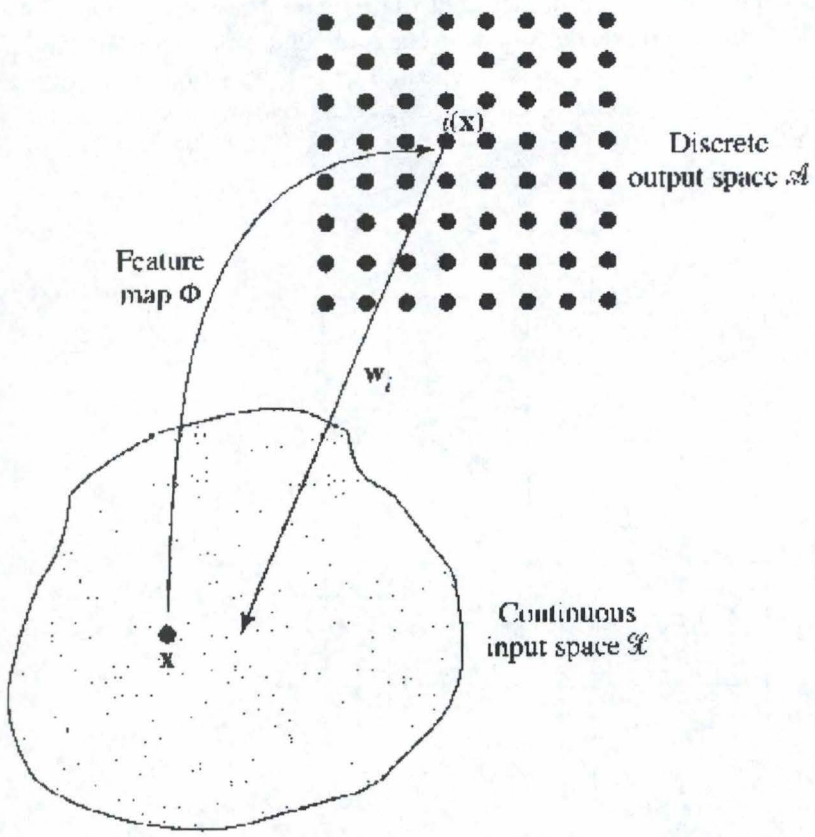
Les positions des neurones « tunés » (c'est-à-dire les neurones gagnants) sont organisées- ordonnées les unes par rapport aux autres, d'une façon qu'un système coordonné et qui a un certain sens par rapport aux différentes caractéristiques de l'input soit créé. Une self-organizing map est, donc, caractérisée par la formation d'une fonction topographique- topographic map- des patrons d'input. Pour cette fonction, les positions spatiales (les coordonnées) des neurones dans le treillis indiquent- désignent- les différentes caractéristiques statistiques contenues dans ces patrons d'input, d'où le nom "self-organizing map" ou SOM.

La SOM, autant que modèle neuronal, offre un pont entre deux niveaux d'adaptation:

- les régles d'adaptation formulées au niveau microscopique d'un seul neurone.
- la formation des patrons qui possède des caractéristiques sélectives- spécifiques-, et qui sont physiquement accessibles et expérimentalement meilleurs; et cela au niveau microscopique des couches de neurones.

Supposons que Φ désigne une transformation non-linéaires qu'on appelle fonction caractéristique, qui transforme (maps) l'espace d'input H en l'espace d'output A de la façon suivante (voir figure suivante):

$$\Phi: H \rightarrow A$$



L'équation de Φ peut être vue comme l'abstraction de l'équation suivante:

$$i(x) = \arg\min_j \| x - w_j \|, \quad j=1,2,\ldots l$$

qui définit la position du neurone gagnant i(x) développé comme réponse au vecteur d'input x. Le symbole: arg $\min_w$ f(w) signifie le minmum de la fonction f(w) par rapport à l'argument qui est le vecteur w, et le symbole ||x|| signifie, bien entendu, la norme du vecteur x.

Nous allons maintenant décrire, sans entrer trop en détails- pour plus de détais cf. [HAY 99]- l'algorithme SOM afin d'illustrer l'ANS, car l'ANS est une auto-organisation comme on l' a déjà vu auparavant. Les ingrédients essentiels utilisés par cet algorithme sont:

- l'input qui est un espace continu de patrons d'activation qui sont générés selon une certaine distribution de probabilité.
- l'output qui est un espace discret définie par la topologie du réseau qui est sous la forme d'un treillis de neurones (voir la figure avant-précédente sur le treillis)
- une fonction de voisinage qui varie avec le temps $h_{j,i(x)}(n)$- n représente le temps ou le nombre d'itérations- et qui est définie autour du neurone gagnant de la façon suivante:

$$h_{j,i(x)}(n) = \exp(-(dj,i)^2 / (2\sigma^2(n))), \quad n=0,1,2,\ldots$$

où $(dj,i)^2 = \| r_j - r_i \|$, r_j est le vecteur- discret- qui définit la position du neurone excité, et r_i la position discrete du neurone gagnant i; tous les deux appartiennent à l'espace discret d'output. σ(n) est définie ainsi:

$$\sigma(n) = \sigma_0 \exp(-n / \tau_1) \quad n=0,1,2,\ldots$$

où σ_0 est la valeur à l'intialisation de l'algorithme SOM, et τ_1 est une constante de temps.

Nous remarquons dans l'équation précédente , que quand le temps n(ou le nombre d'itérations) augmente ou découle, l'ampleur de σ(n) décroit avec un taux exponentiel, et le voisinage topologique se rétrécit de manière correspondante, et pour cela, d'ailleurs, on appelle $h_{j,i(x)}(n)$ la fonction du voisinage.

- un paramètre du taux d'apprentissage $\eta(n)$ qui commence avec une valeur initiale η_0 et puis décroit graduellement, mais n'atteint jamais 0.

$$\eta(n) = \eta_0 \exp(-n / \tau_2) \quad n=0,1,2,...$$

Où τ_2 est une autre constante de temps de l'algorithme SOM.Nous remarquons le décroissement exponentiel avec le temps de $\eta(n)$.

L'algorithme SOM, en résumé, se déroule de la façon suivante:

1. **Initialisation-** Choisir aléatoirement les valeurs des poids $w_j(0)$. La seule restriction ici est que les $w_j(0)$.doivent être différents pour j=1,2,...*l* où *l* est le nombre de neurones dans le treillis.Il est peut-être souhaitable de garder les valeurs des poids petits. Une autre façon d'initialiser l'algorithme est de sélectionner aléatoirement les vecteurs des poids{ $w_j(0)$}, j=1,..*l* parmi l'ensemble de vecteurs d'input disponibles {xi}, i=1,..,N.

2.**Echantillonnage**.
Tirer, avec une certaine probabilité, un échantillon x de l'espace d'input; le vecteur x représente le patron d'activation qui est appliqué au treillis. La dimension du vecteur x est m.

3.**L'appareillement** (similarity matching). Trouver le neurone qui a le meilleur appareillement(le gagnant) i(x) au temps n- ou à l'itération n- en utilisant la distance euclidienne minimum:

$$i(x) = \arg \min_j \| x - w_j \|, \quad j=1,2,...l$$

4. **La mise à jour**. Ajuster les vecteurs de poids synaptique de tous les neurones en utilisant la formule suivante:

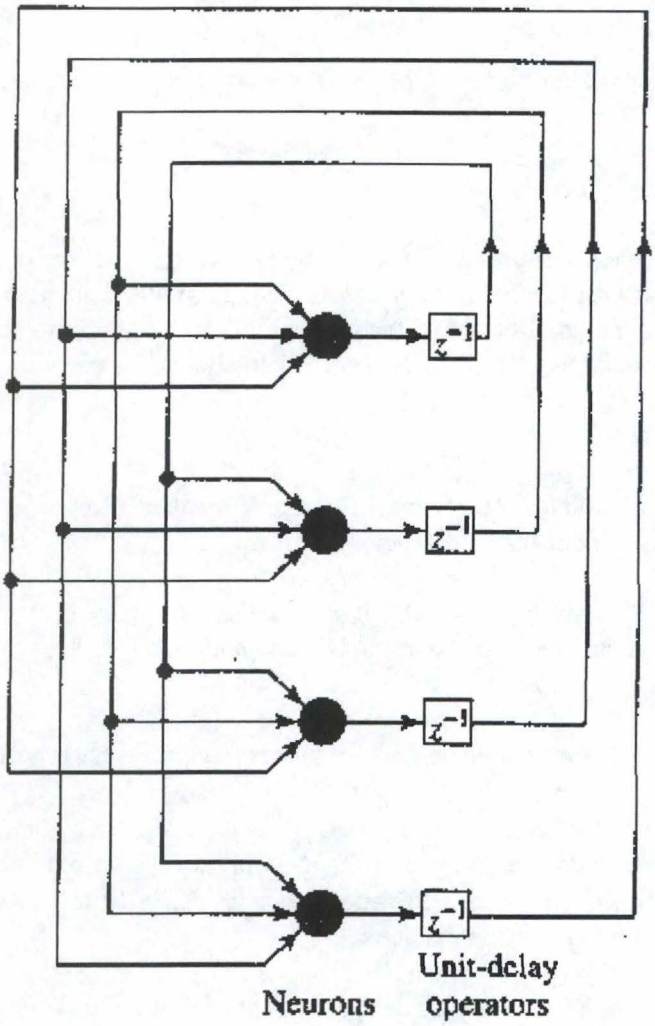
$$w_j(n+1) = w_j(n) + \eta(n)\, h_{j,i(x)}(n)(x(n) - w_j(n))$$

où, comme on l' déjà cité, $\eta(n)$ est le paramètre du taux d'apprentissage, et $h_{j,i(x)}(n)$ est la fonction du voisinage centrée autour du neurone gagnant i(x); $\eta(n)$ et $h_{j,i(x)}(n)$ varient, tous les deux, dynamiquement durant la période d'apprentissage afin d'obtenir de meilleurs résultats.

5. **Continuation**. Continuer avec l'étape 2 jusqu'à qu'il y a plus de changements sensibles- significatifs- observés dans la fonction- dans sa valeur-.

Après que nous avons étudier un exemple de l'ANS basé sur le processus compétitif, nous allons maintenant étudier un autre exemple pratique de l'ANS, basé lui sur les processus adaptatif, qu'on appelle les réseaux de Hopfield.

Le réseau de Hopfield conceptualise le réseau neuronal en termes d'énergie et de la physique des systèmes dynamiques. Il est totalement connecté, symétrique(la matrice des poids est symétrique $w_{ji} = w_{ij}$ pout tout i et j), multi-rétroactifs (feedback), et les neurones ont des valeurs binaires(0 ou1). (voir la figure suivante).



Etant donné un ensemble d'inputs, les poids du réseau- plus exactement des neurones du réseau- sont mis à jour d'une façon répétitive par un algorithme simple local, et nous aurions comme résultat que le réseau se relaxe dans un état qui représente la mémorisation (la valeur stockée en mémoire). Cet état existe (se trouve) dans un minimum local de la fonction d'énergie (la fonction de Lyapunov) qui par définition est non-croissante, et pour cela, d'ailleurs,on l'utilise pour les mises à jour de l'état du réseau. La valeur de l'énergie calculée par cette fonction représente le degré de non-optimalité de l'état courant du réseau.

Comme il est illusté par la figure précédente, il y a autant d'unités de retardement (unit delay) que de neurones, ainsi l'output de chaque neurone est remis rétroactivement, via une unité de retardement, à chacun des autres neurones du réseau. La mise à jour se fait un neurone à la fois (séquentiellement) et d'une façon asynchrone. L'activation de chaque neurone est non-linéaire; les poids positifs des neurones sont excitants et vont renforcer les connexions, tandis que les poids négatifs sont inhibitatoires et affaiblissent les connexions.

Vu que la fonction (de Lyapunov) de l'énergie du réseau de Hopfield est monotone, et décroissant avec le temps, alors le réseau est *globalement, et asymptotiquement* stable ; les attracteurs des points fixes sont les minima de la fonction d'énergie, et vice versa.

Le réseau a des fortes limitations en ce qui concerne la capacité de sa mémoire. Dans sa forme originale, la capacité maximum de la mémoire est 0.1N (où N est le nombre de neurones dans le réseau). Le dépassement de cette limite produit des faux états avec des outputs incorrects. On appelle cela le phénomène du "déjà vu"; le réseau se rappelle des états qu'il les a jamais appris. Nous allons à présent décrire succinctement le processus adaptatif d'apprentissage (non-supervisé) de ce réseau.

Procédure d'adaptation

- initiallement, on assigne un état à chaque neurone du réseau et on adapte les neurones un à la fois. La procédure d'adaptation est appliquée aux neurones jusque quand elle n'effectue plus aucun changement au réseau .
- chaque neurone calcule la somme de ses inputs :

  $S_j = \Sigma^{n}_{i=0} a_i w_{ji}$

  avec a_i la valeur d'input de l'unité i

  w_{ji} le poids associé à la connection entre l'unité i et l'unité j
- le neurone calcule son état à partir de cette somme :

  if $S_j > 0$ then $u_j = 1$

  if $S_j \leq 0$ then $u_j = 0$

  avec u_j la valeur d'output de l'unité j
- l'ancienne valeur d'état du neurone n'est pas prise en compte lors de l'adaptation . L'état du neurone peut changer mais peut rester identique .
- l'adaptation des neurones est asynchrone (pas tous les neurones d'une couche en même temps) → plus ressemblant au neurone biologique .

Convergence

- chaque état d'un réseau Hopfield est caractérisé par une valeur : son "énergie" :

  $E = -1/2(\Sigma_j \Sigma_i w_{ji} u_j u_i)$

Cette fonction d'énergie est minimisée par le réseau.

2.7.4. La combinaison de RN et AG

- **Introduction**

L'idée de combiner lès AG et les RN (ou les NN) est apparue vers la fin des années '80, et a mené à de nombreuses recherches depuis lors. Comme les deux sont des méthodes de programmation autonomes, pourquoi les combiner ?

Le problème avec les NN est qu'un certain nombre de paramètres doivent être définis avant de commencer l'apprentissage. Cependant, il n'existe pas de règles précises sur la façon de définir ces paramètres. Pourtant ce sont eux qui déterminent le succès de l'apprentissage.

En combinant les AG avec les NN (GANN), l'AG est utilisé pour trouver ces paramètres . L'inspiration de cette idée provient de la nature : dans la vie réelle, le succès d'un individu n'est pas uniquement déterminé par ses connaissances et aptitudes, qu'il a obtenues par son expérience (l'apprentissage du réseau de neurones), cela dépend aussi de son héritage génétique (déterminé par l'AG). On peut dire que GANN applique un algorithme naturel qui a fait ses preuves sur cette planète : il a créé l'intelligence humaine à partir de rien.

- **Idée Générale**

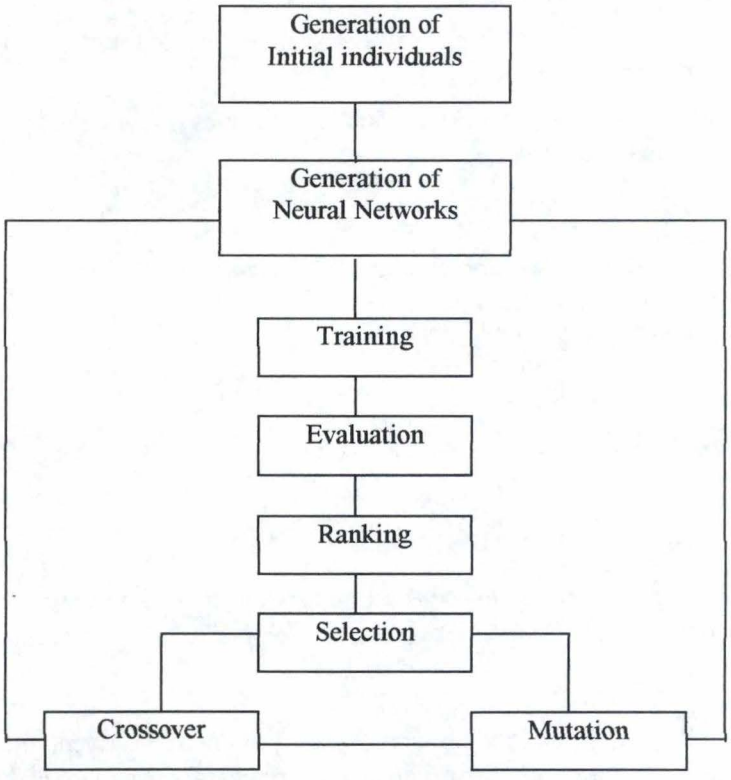
L'idée générale du GANN est illustré par le schéma ci-dessous.
L'information du NN est encodée dans l'ADN (le chromosome) de l'AG. Au départ, un certain nombre d'individus aléatoires sont générés. Les strings binaires doivent être évalués, ce qui signifie qu'un NN doit être créé selon l'information du chromosome.

Sa performance peut être déterminée après l'apprentissage par la propagation vers l'arrière. Cependant, certaines stratégies GANN ne comptent que sur le AG pour trouver un réseau optimal ; dans celles-ci, aucun apprentissage n'a lieu.

Ensuite, ils sont évalués et classés. L'évaluation du fitness peut prendre en considération plus de paramètres que la performance de l'individu. Certaines approches prennent en compte la taille de réseau afin de générer les plus petits réseaux possibles.

Enfin, le crossover et la mutation créent de nouveaux individus qui remplacent les moins bons ou tous les membres de la population.

Cette procédure générale est en réalité assez simple. Le problème du GANN, cependant, se situe dans l'encodage du réseau.



La structure principale d'un système GANN

- **Encodage du réseau**

Il existe de multiples encodages possibles. Les deux grandes familles d'encodages **directs** et **indirects**.

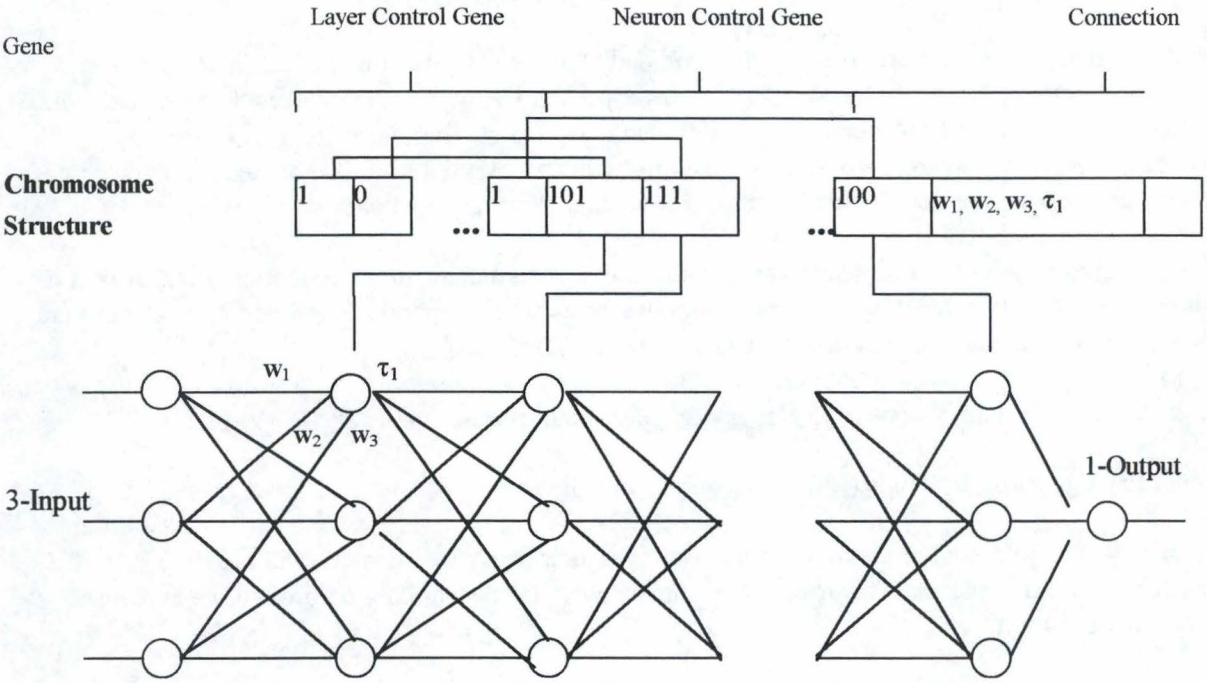
L'encodage direct fait référence aux qui encodent directement les paramètres du réseau de neurones tels que les poids des synapses, la forme des connections, etc. dans le chromosome. A titre d'exemple, le poids des synapses peut être encodé soit sous forme binaire soit sous forme décimale.

A l'opposé, l'encodage indirect utilise l'encodage de règles qui contiennent les informations concernant la construction du réseau.

De nombreuses études ont été réalisées sur la façon d'encoder un réseau de neurones le plus efficacement possible. Examiner en détail chacune d'elle nous emmènerait au-delà de l'objectif de ce travail. Toutefois, citons brièvement deux d'entre elles :

D'abord, la première étude a montré que l'utilisation des AG pour remplacer la propagation vers l'arrière en vue de l'optimisation des poids ne semble pas être très compétitive, surtout où la vitesse du calcul -computational speed- est un paramètre important. Cependant, cette utilisation peut être une méthode d'apprentissage prometteuse en vue de renforcer l'entraînement d'un réseau (NN) à récurrence totale -full recurrent-, ou d'un réseau où les neurones utilisent le mode de transfert - propagation- non-différentiable (non-differentiable transfert).

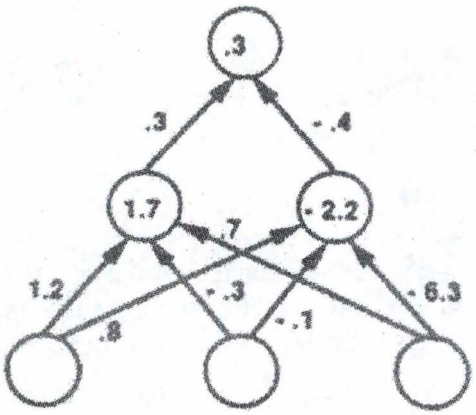
La deuxième étude montre que la contribution des AG à l'optimisation de la topologie globale d'un NN est faisable et réalisable. Une application typique des AG est de faire évoluer la topologie du réseau en utilisant les AG pour l'ajustement fin (fine tuning) des poids. Cette approche a été démontré - et réalisée - dans les réseaux hiérarchiques[7] à propagation vers l'avant, où on a optimisé globalement et la topologie et les coefficients des poids. La structure des chromosomes utilisée dans cette approche est illustrée dans la figure suivante :



n ce qui concerne l'utilisation des AG comme méthode prometteuse d'apprentissage des NN, on peut faire les remarques suivantes :

1) Le codage : On peut coder, par exemple, les poids (et les biais) du NN par une liste ordonnée selon un ordre bien déterminé, comme il est illustré par la figure suivante :

[7] Ce sont des réseaux à multicouche (ou multi-niveau) agencés hiéarchiquement, de façon que caque couche correspond à un niveau - ou subdivision - fonctionnel ou décisionnel.



Le codage ⇒ (0.3,-0.4,0.3,1.2,0.8,-0.3,-0.1,0.7,.-6.3,1.7,-2.2)
Le codage du NN par un chromosome

2) La fonction d'évaluation (ou de gain) : c'est la somme des carrés des erreurs à chaque cycle d'apprentissage du réseau sur un patron ou un exemple. L'erreur(δ) pour chaque unité étant la différence entre la valeur désirée et la valeur effective de cette unité.

3) Initialisation : Les inputs (les poids) des individus de la population initiale sont générés aléatoirement- comme d'habitude- avec une certaine probabilité de distribution, comme par exemple, $e^{|x||}$,qui reflète l'observation empirique. Ainsi, cette population initiale et les générations suivantes seront pour l'AG l'espace de recherche de touts les solutions possibles. L'AG va explorer toutes ces solutions, et va converger vers les meilleures- au sens de la fonction d'évaluation- de ces solutions.

4) Les opérateurs : Ce sont les mêmes opérateurs génétiques habituels de reproduction, crossover et de mutation, mais qu'on a créés spécialement pour les besoins de l'apprentissage des NN, et qui sont adaptés pour ce genre des réseaux (pour plus de détails cf. [DAV 91]).

Pour ce genre de réseaux. Citons en quelques exemples (pour plus de détails cf. [DAV91]).

A) Mutation Unbiaisée des Poids (Unbiased-Mutate-Weights):

Pour chaque entrée(poids) du chromosome, cet opérateur va, avec une probabilité fixe p = 0.1, remplacer cette entrée par une valeur aléatoire choisie avec une distribution de probabilité la même que celle de l'initialisation.

B) Mutation des noeuds (Mutate-Nodes):

Cet opérateur sélectionne N noeuds non-inputs (n'appartiennent pas à la couche d'entrée) du réseau que le chromosome parent représente. Pour chaque lien (synapse) entrant dans ces N noeuds, l'opérateur ajoute au poids du lien une valeur aléatoire choisie avec la distribution de probabilités de l'initialisation. Puis, il encode ce nouveau réseau dans le chromosome de l'enfant (descendant).

L'idée ici est que les liens entrants dans un noeud forment (logiquement ou conceptuellement) un sous-groupe de tous les liens par rapport à l'opération (la fonction) du réseau. En limitant les changements du réseau à un petit nombre de ces sous-groupes, cela va faire que les améliorations vont -plus probablement- donner comme résultat une meilleure évaluation (une meilleure convergence vers le patron recherché).

C) Crossover des Poids (Crossover-Weights):

Cet opérateur met une valeur dans chaque position du chromosome de l'enfant, en choisissant aléatoirement un des deux parents, et en utilisant la valeur de la même position dans le chromosome de ce parent.

D) Crossover des Noeuds (Crossover-Nodes):

Pour chaque noeud du réseau encodé dans le chromosome de l'enfant, l'opérateur sélectionne un des deux réseaux-parents (en fait, on génére aléatoirement les individus à l'initialisation, cf.3) Initialisation), puis il trouve le noeud correspondant dans le réseau-parent. Alors, il met les poids de chaque lien entrant dans le noeud-parent dans le lien correspondant dans le réseau-enfant.

L'idée ici est que les réseaux réussissent grâce à la synergie entre leurs différents poids, et cette synergie est plus grande entre les poids des liens (synapses) entrants au même noeud. Par conséquent, comme le matériel génétique peut contourner cette synergie, alors on doit garder ces sous-groupes-conceptuels-ensemble.

5) Les paramètres : Ils sont les mêmes que pour les autres applications des AG, donc la taille de la population, le nombre de générations, P_c , P_m , élitisme, windowing et fitness scaling (pour les définitions de ces dernières cf. §3.2.).

- **Exemple d'application**

Pour illustrer l'idée du GANN, nous allons décrire une "nouvelle" méthode d'apprentissage des NN basée sur cette idée.Cette nouvelle approche s'appelle 2DELTA-GANN (cf. [KC 94]).

1) Introduction

Nous présentons une méthode de l'utilisation des AG dans l'apprentissage des réseaux de type perceptron multicouches, dans lesuqels les chromosomes encodent des "règles" pour le changement des poids du réseau, plutôt que encoder les poids eux-mêmes. Les opérateurs génétiques de sélection, du crossover et de mutation sont utilisés pour générer des règles nouvelles qui seront, à leur tour, appliquées à la matrice des poids.

Cette approche est significativement meilleure- comme on va le voir- que les autres approches pour l'apprentissage des réseaux en utilisant les AG, et elle résout avec succès un bon nombre de problèmes de référence- benchmark problems- qui sont connus pour être difficilles pour la méthode de la propagation de l'erreur vers l'arrière (BP). En effet, la détermination des poids d'un perceptron multi-couches est essentiellement un problème de recherche multi-dimensionel (non-linéaire). La BP et ses variantes sont, pour le moment, les méthodes les plus utilisées pour la recherche d'un pint optimal dans l'espace des poids. Cependant, les AG fournissent une méthode alternative pour l'implémentation de cette recherche.

L'approche avec les AG implique: l'encodage des solutions potentielles par des chaînes de bits- des chromosomes-, la création de la population initiale de chromosomes et l'utilisation des opérateurs génétiques (la sélection, le crossover et la mutation) pour dériver la nouvelle population. Comme nous l'avons déjà vu, théoriquement ces opérateurs sélectionnent les meilleures parmi les solutions potentielles de la génération courante- actuelle-, puis ils les combinent pour former une nouvelle population qui devrait être meilleure que la précédente. La reproduction d'une nouvelle génération

continue jusqu'à que la meilleure solution soit "assez bonne" par rapprot à un certain critère. Cette procédure exige une fonction de fitness qui retourne une valeur numérique pour chaque chromosome.

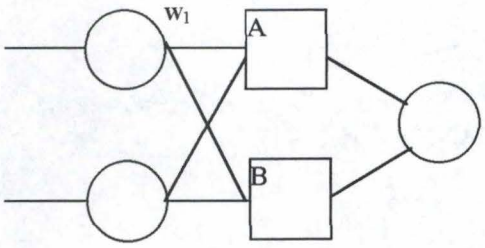
Afin d'utiliser la reherche génétique- par un AG- pour la détermination de l'ensemble optimal des poids d'un réseau, nous avons représenté les poids et les biais par un chromosome. La position qui encode un seul poids correspond à un gène. Chaque chromosome représente l'ensemble total des poids d'un réseau, et il est en même temps un individu d'une population. Les membres de la population sont sujets aux actions des opérateurs génétiques de sélection, de crossover et de mutation.

Après qu'on a calculé et déterminé les chromosomes de chaque nouvelle génération, les poids du réseau sont déterminés à partir de chaque chromosome, et l'erreur par rapport à l'ensemble d'apprentissage est calculée. La mesure de l'erreur est utilisée comme une mesure de la fitness du chromosome.

Dans les travaux précédents sur l'utilisation des AG dans l'apprentissage des NN, les gènes des chromosomes encodaient directement les poids du réseau, tandis que dans l'approche que nous présentons ici, un gène représente une "règle" ou une méthode utilisée pour changer le poids correspondant.

2) La méthode

Les différents aspects de la méhode- de la nouvelle approche- sont décrits dans ce qui suit, en se référant au réseau XOR de la figure suivante:



2.a) La structure du gène

Dans notre approche nous avons utilisé l'encodage indirect,et Le gène est une structure composée de:

- 3 bits pour la règle qu'on les nomme x_1, x_2 et x_3.
- 2 valeurs en virgule flotante qu'on les nomme $delta_1$ et $delta_2$.

Les valeurs de x_1, x_2 et x_3 spécifient une règle heuristique qu'on applique à $delta_2$. $delta_2$ va alors modifier $delta_1$, qui sera, à son tour, appliquée aux poids et biais associés avec le gène en question. Les bits de la règle sont interprétés de la façon suivante:

```
if x1 then
  if x2 & x3 then
     delta2 *= 2
  else
     delta2 /= 2
  delta1 += delta2
  endif
endif
weight( ou biais) += delta1
end
```

Par exemple, si les bits x_1, x_2 et x_3 sont tous à 1, alors $delta_2$ est doublé, le nouveau $delta_2$ est ajouté à $delta_1$, et le nouveau $delta_1$ est ajouté- à son tour- au poids w_1.

Le mécanisme du changement permet qu'un poids soit changer par une valeur de $delta_1$, qui à son tour, peut être modifié par la valeur de $delta_2$. Le taux du changement du poids et du $delta_1$ (leurs "gradients") est changeable grâce à la valeur de $delta_2$. Celui-ci est utilisé pour fournir une heuristique "deuxième ordre" ou "deuxième niveau" un mécanisme de la règle de modification des poids. Cette utilisation des delta-valeurs est comparable à "l'encodage delta" décrit en [WHI 91].

Notons que la longueur du chromosome qui devient de l'ordre de centaines ou de milliers de bits quand utilise des mécanismes comme l'encodage dynamique des paramètres [SCH 90], sera plus courte quand on utilise cette représentation (l'encodage indirect de cas ici). Le fait de manipuler des chromosomes plus courts nous permet de pouvoir résoudre, en utilisant les techniques génétiques, des problèmes plus grands- dont les chromosomes sont plus petits-.

2.b) La sélection

Nous utilisons la technique la plus répandue et la plus utilisée qu'est "la roue de loterie"(cf.§2.3.3.). Cependant, les opérateurs de crossover et de mutation doivent être modifiés pour s'adapter à la nouvelle structure du gène.

2.c) Le crossover

L'opérateur du crossover est basé sur le crossover uniforme(cf.§2.3.4.). Le masque du crossover est appliqué seulement aux bits de règles du chromosome, et pas aux deltas. Si il y a eu un échange- ou une permutation- parmi n'importe quels bits de règles pendant le crossover, alors les valeurs de $delta_1$ et $delta_2$ sont échangées entre les chromosomes des parents.

Nous avons choisi le crossover uniforme pour plusieurs raisons. Premièrement, pour sa capacité de combiner les blocs de constructions(cf.§2.3.4.) du matériel génétique dispersés dans les chromosomes longs (comme il était démontré par Syswerda, cf. [SYS 89]) ; mais aussi pour le fait que le crossover uniforme peut rendre l'inversion ou la mutation, autant qu'un opérateur de réarrangement, inutile. Dans le contexte du problème d'optimisation des poids d'un réseau neuronal, cela devient très important. Deuxièmement, si un réseau doit être entraîné sur une partie de l'espace de solutions par des changements des poids qui sont fort dispersés sur le chromosome, la liaison ou les liens entre les poids sera, vraisemblablement, moins établie par des autres types de crossover comme, par exemple, le crossover de 1-point ou de 2-points. Troisièmement, comme il est cité dans la littérature spécialisée, le crossover uniforme est un meilleur explorateur, probablement, que le crossover classique. Ceci est important dans l'exploration des espaces multimodaux qu'on rencontre souvent dans les problèmes d'optimisation des réseaux neuronaux.

2.d) La mutation

quand la mutation est appliquée, alors elle l'est uniquement sur les bits des règles, et ainsi des nouvelles valeurs de $delta_1$ et de $delta_2$ sont générées aléatoirement, et elles sont utilisées pour ce gène(le gène qui a subi la mutation).

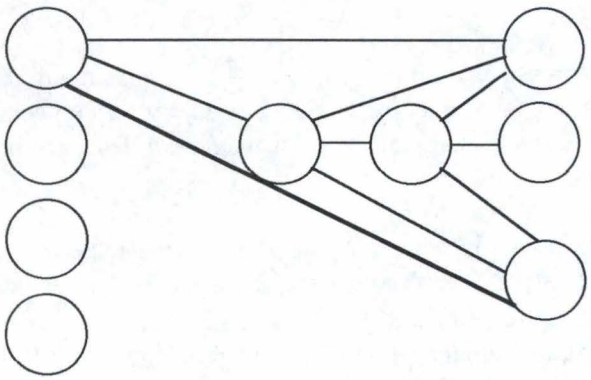
Quand on a utilisé les AG à état stable (cf.§2.3.3.), on a aussi utilisé l'opérateur de mutation adaptative de Whitley (cf. [WHI 89b]). Cet opérateur applique la mutation aux descendants issus du crossover, avec un niveau- ou un taux- proportionel à la différence- mesurée d'une certaine façon- enrte les deux parents. Cependant, nous avons utilisé, dans notre cas, une simplification de cet opérateur en mesurant, tout simplement, la différence entre les deux parents en termes de différence de fitness entre les deux

chromosomes- des deux parents bien évidement-. Aucune méthode spéciale n'a été utilisée pour mesurer la différence entre deux chromosomes, malgré le fait que ces chromosomes ont, dans notre cas, une structure spéciale de leurs gènes.

2.e) La topologie du réseau

En plus de changements que nous avons opéré concernant la structure du chromosome et le mécanisme du changement des poids, nous avons procédé, de plus, à une modification- standard- du réseau neuronal, afin d'éviter le problème de "conventions de la concurrence" (cf. [SCH 92]). Ce problème peut être décrit- brièvement de la façon suivante:
Dans le réseau XOR (voir figure précédente, point 2), si on échange- on permute- les poids et les biais de l'input et de l'output des unités (des noeuds) A et B, le réseau continuera à calculer et à sortir le même output- final bien entendu-. Pour un algorithme génétique, cela pose un problème (considéré par Belew (cf. [BEL 90]), d'ailleurs, comme un problème dévastateur), car les deux réseaux (avant et après la permutation) ont différents chromosomes, mais ils ont la même fitness. Ainsi la population de chromosomes peut aboutir à une diversité apparente qui, en réalité, est beaucoup moins de ce qu'elle aparaît, ce qui conduit à une convergence prématurée de l'AG.

Pour éviter ce problème, nous utilisons uniquement une nuité- un noeud- par couche cachée, et on conncete chaque couche cachée à toutes les couches qui la précèdent. La figure suivante montre le réseau du porblème "additionneur minimal 423" (4 inputs, 2 cachées, 3 outputs).



La topologie du réseau de l'additionneur minimal 423
(uniquement les connections à partir d'une unité d'input sont illustrées)

Cette façon de construire le réseau donne une structure de réseau en cascade, similaire au réseau de Fahlman (cf. [FAH 90]) basé sue l'architecture de corrélation en cascade. En pratique, les réseaux avec ce type de "cascade" sont optimisés beaucoup lpus vite que les réseaux habituels qui contiennent plusieurs unités par couche cachée, et où il n'y a pas d'interconnexions dans une couche.

3) Implémentation

Nous avons utilisé dans les expériences que nous avons mené ici 2 logiciels spécialisés forts répandus basés sur 2 variantes des AG:
1) le remplacement de génération- generational replacement-(Genesis)
2) l'état stable (Genitor)

Dans le remplacement de génération, un espace est alloué pour les deux populations. La nouvelle population est calculée uniquement à partir de l'ancienne. Pour ces expériences nous avons utilisé le programme Gaucsd 1.4, avec des modefications pour savoir manipuler la nouvelle structure du gène.

Dans l'approche de l'état stable, un espace est alloué pour une seule population. Les nouveaux enfants remplacent- avec écrasement- les membres existants de la population. pour ces expériences, nous avons utilisé l'algorithme de Whitley (cf. [WHI 89a]) GENITOR, après l'avoir modifié, lui aussi, pour convenir à la nouvelle structure du gène. Le logiciel du NN utilisé ici était essentiellement une version modifiée du programme "bp" qui est disponible avec le livre du PDP (cf. [RUM 89]).

4) Les résultats

Les résultats du tournage- de l'exécution- de l'algorithme sur un nombre de problèmes sélectionnés sont illustrés dans le tableau suivant:

No.	Problem	2DELTA-GANN				Whitley-GANN		BP
		Regime	Trials	Pop	Success	Trials	Pop	Trials
1	10-5-10 Encoder	Genesis	15,100	1000	100%	NA		258
2	10-5-10 Encoder	Genitor	5,800	200	75%	NA		258
3	10-5-10 Comp	Genesis	168,300	1000	42%	NA		668
4	10-10-1 Dtoa	Genesis	22,210	1000	95%	NA		248
5	10-10-10 Random	Genesis	207,430	1000	70%	NA		98
6	6-6-1 Parity	Genesis	912,000	1000	25%	NA		
6a	6-6-1 Parity	Genesis	112,000	1000				127,000
7	6-6-1 Parity	Genitor	137,200	300	28%	NA		
7a	6-6-1 Parity	Genitor	37,200	300				127,000
8	4-2-3 Minimal adder	Genesis	130,430	1000	56%	100,000	3,000	Fails
9	4-4-3 Adder	Genesis	63,500	1000	56%	1,250,000	2,000	38,600
10	4-4-3 Adder	Genitor	33,000	100	50%	500,00	5,000	38,600
10a	4-4-3 Adder	Genitor	9,000	100				

Les problèmes 1-5 sont basés sur un ensemble utilisé par Veitch (cf. [VEI 91]) pour mesurer- benchmarcking- les algorithmes d'apprentissage des NN, et ils montrent la performance de 2DELTA-GANN comparée à la propagation vers l'arrière-BP-. Dans les deux cas, on a pris la moyenne de résultats sur 25 essais. Dans tous les problèmes où l'output est 0/1, 2DELTA-GANN s'arrêtait quand toutes les valeurs de l'output pour les buts"1" étaient 0.6 (60%) ou plus, et les valeurs de l'output pour des buts "0" étaient 0.4 ou moins comme chez [VEI 91]. Pour les deux approches génétiques précitées et reprises dans le tableau, un "essai" correspond exactement à l'évaluation de la fonction de fitness. Donc, 1 essai implique un passage en arrière à travers l'ensemble de l'apprentissage. Il n'y a pas d'équivalent d'un "essai" en BP. Cependant, ils existent une propagation vers l'arrière et une propagation vers vers l'avant à traves le réseau pour chaque entraînement par époque- génération-. En conséquence, une époque correspond exactement à 2 essais. Cette méthode est utilisée dans la détermination du nombre "d'essais" pour la colonne BP.

Nous allons décrire avec un peu plus de détails les résultats du tableau suivant l'ordre de NO:

1) c'est le problème standard de l'encodeur 10-5-10. En utilisant le régime Genesis, 2DELTA-GANN résout ce problème en 15100 essais. La taille de population utilisée était 1000, et toutes les tentatives ont convergé vers une solution. BP avait besoin de 258 essais pour ce problème.

2) c'est le même problème que en 1, mais on a utilisé le régime du contrôle Genitor.

3) ceci est l'encodeur à complément 10-5-10(cf.[VEI 91]).

4) c'est le problème de "numérique/analogique" (digital to analog) chez [VEI 91]. Un nombre binaire de 10 chiffres (10 bits) contient un seul bit à 1, et il faut le transformer- to be mapped- à un output en virgule flotante de la manière suivante:
1000000000→0.1, 0100000000→0.2, 0010000000→0.3,...le seuil de l'erreur est 0.03 pour chaque output. Si le but-output est- par exemple, 0.4, alors l'output du réseau est considéré correct seulement si il est dans l'intervalle 0.37 à 0.43. Ce problèem teste "l'apprentissage fin ou précis".

5) ce problème exige du réseau l'apprentissage d'une transformation- mapping- entre 10 inputs aléatoires et 10 outputs aléatoires.

6-7a) ce réseau est utilisé pour la solution du problème de la parité de 6 bits. 6a et 7a donnent les résultats pour ce problème seulement dans les cas où les exécutions covergent. Plus que la moitié des exécutions de BP échouent pour ce problème.

8) ce problème exige l'addition de deux nombres, chacun a 2 bits, et fournit la somme en 2 bits avec un bit de report. Le réseau minmum pour ce problème possède 2 noeuds cachés et des connexions directes entre l'input et l'output. BP devient "clouée"- se cale- dans un minimum local de ce réseau. Des noeuds cachés supplémentaires sont nécessaires (cf.[RUM 98]). Les résultats de GANN de Whitley sont tirés de [WHI 89b]. Cependant, les résultats de deux GANN ne sont pas directement comparables, car on a utilisé différents critères de convergence. Le GANN de Whitley avait terminé avant que l'erreur était réduit au niveau désiré.

9-10a) c'est le problème de l'additionneur à 2 bits avec 4 unités dans la couche cachée. les résultats de GANNde Whitley sont tirés de [WHI 89a], [WHI 89b]. Pour les petits problèmes, BP est clairement supérieur aux deux variantes de 2DELTA-GANN. Cependant, ces problèmes d'apprentissage sont très simples, et ne posent pas de problèmes pour les méthodes standard d'apprentissage. Afin de comparer convenablement les méthodes d'AG avec des méthodes comme la BP, il faut que les problèmes soient multimodaux et difficiles à résoudre.

Avec des simples problèmes unimodaux, on ne peut pas s'attendre à que les AG surpassent les performances de la méthode de descente de gradient. Cependant, quand on veut résoudre des problèmes plus difficiles comme le problème de parité de 6 bits, 2DELTA-GANN devient compétitive avec BP et commence même à la dépasser.

2DELTA-GANN est clairement supérieure au GANN de Whitley qui encode les poids du nréseau directement dans le chromosome. Nos résultats sur l'AG(état stable-Genitor) comparé à celui de remplacement de génération(Genesis) sont en accord avec ([WHI 89b]). Lapproche de l'état stable est clairement supérieure.

5) Conclusions

Nous avons présenté une méthode, 2DELTA-GANN, d'apprentissage du NN par l'AG. La méthode est apparemment supérieure aux autres algorithmes GANN de la littérature, en termes de nombre d'essais pour la convergence, et en même temps, en longueur du chromosome.

Pour les petits problèmes, BP est meilleure que 2DELTA-GANN. Cependant, 2DELTA-GANN résout quelques problèmes connus pour être difficiles pour BP. Des études supplémentaires sont nécessaires pour tester 2DELTA-GANN sur des problèmes difficiles.

Un avantage de 2DELTA-GANN est le fait q'elle n'a pas besoin d'information sur le gradient. Cela veut dire qu'on peut faire des expériences sue le NN avec différents types de fonctions d'activations. Un autre avantage de 2DELTA-GANN est qu'on peut l'exécuter- la réaliser- quand l'AG est exécuté sur

des machines massivement parallèles. La nature parallèle de l'AG signifie que en exécutant les programmes sur de telles machines, on espère d'augmenter la vitesse d'exécution par des facteurs égaux à la taille de la population, car on peut appliquer des "individus"- ou des exemplaires- de la population de l'AG sur des copies séparées- et en même temps- de réseaux neuronaux, et les évaluer en parallèle. Sur de telles machines on peut s'attendre à que l'AG- plus exactement 2DELTA-GANN- surpasse confortablement les performances de BP.

Chapitre 3: La boîte noire

3.1. Introduction

Comme toute action a des limites dans l'espace et dans le temps, notre mémoire est limité dans sa portée et malheureusement l'espace disponible qu'il nous reste pour notre exposition est restreint.
En fait cette contrainte nous a accompagné tout au long de ce mémoire, ce qui nous a obligé à plusieurs reprises à limiter l'exposé dans son étendue et dans sa profondeur, mais c'est ainsi l'ordre des choses.

Pour cette raison impérative, nous avons exposé dans ce qu'il nous restait d'espace, succintement et brièvement, la définition, la configuration, l'implémentation et la conclusion de l'application - la boîte noire en l'occurence-. Pour plus de détails et d'informations, on peut se référer à l'annexe, où on a décrit, avec plus de détails, la fonction sous-jacente de la boîte noire, l'implémentation de celle-ci, les résultats numériques des exécutions de l'AG et des graphiques illustrant les influences des paramètres.

3.2. Définition

Il s'agit d'un objet - réel ou virtuel- qui à chaque valeur d'entrée fournit - retourne ou renvoie- une valeur de sortie. Donc, pour chaque signal d'entrée **s** , il y a un signal de sortie **f** ;
mathématiquement **f = f (s)** . Personne sait ce qu'il se passe à l'intérieur de la boîte, mais on connaît, seulement, les faits suivants :

- L'input de la boîte est un entier signé codé sur 32 bits.
- L'output de la boîte est un entier signé de 32 bits, dont la valeur de cet entier est entre 0 et 32 .
- Un seul input - bien sûr inconnu, sinon où est la magie de la boîte noire et la puissance des AG- a comme output la valeur 32 .
- Les valeurs inférieurs à 32 peuvent être produites - générées - par plusieurs inputs (autrement dit le même output peut correspondre à plusieurs inputs .
- Il n'existe pas une correspondance - ou une relation - évidente entre les valeurs de l'input et leurs outputs . Par exemple, l'input 32 - l'entier dont la valeur est 32 - produit un output 0, tandis que un input = 10 produit output = 4 .
- On ne peut pas ouvrir la boîte, ni effectuer des autres investigations sur elle; on peut seulement, fournir des en input et examiner - ou connaître - les valeurs associées - correspondantes - en output.

Notre tâche, ou plus exactement **l'objectif du problème, est double** :
1) d'abord, **trouver la valeur d'input** - une seule en l'occurrence - qui produit l'output égal à 32 ; autrement dit, on veut maximiser l'output de la boîte noire - car 32 est la plus grande valeur possible de f
2) et puis **tester les influences des paramètres de l'AG** sur le nombre de générations (ou le temps d'exécution qui est ici en tick) avant la convergence de l'AG vers la solution recherchée, autrement dit la vitesse de convergence de l'AG.

Ici, il y a lieu de faire plusieurs remarques très importantes :

A) D'abord, avec cette boîte on peut simuler beaucoup de choses : un réseau de neurones, un circuit électrique ou électronique, un circuit logique ou combinatoire, un processus de décision, ou en général,

n'importe quelle fonction mathématique (cf.[MAW 84], [APO 82]), peu importe sa forme ou son expression analytique. Cela donne à ce problème - ou à cette boîte - une très grande généralité, et en même temps, montre la grande puissance des AG dans des domaines tels que l'optimisation, la décision,...etc.

B) On verra, grâce à ce problème, que les AG, n'ont besoins que d'un codage convenable - ou adapté au problème- et d'une mesure des gains. La connaissance du fonctionnement de la boîte noire ne leur est pas nécessaire, donc peu importe l'expression de la fonction, et d'ailleurs pour cette raison on ne propose pas aux utilisateurs de ce logiciel d'entrer la fonction sous-jacente à cette boîte (cf.Annexe,6.Configuration et implémentation, Remarques sur la fonction d'évaluation) comme on le fait pour les autres paramètres.

Pour les besoins de cet exemple, on a défini une fonction *Blackbox* qui a le prototype suivant :

```
long    Blackbox ( long  x ) ;
```

L'utilisateur n'a pas aucune idée sur l'implémentation de Blackbox ; ca se peut, par exemple, qu'elle est définie dans un module dont a perdu le code-source.

Une approche avec une force brutale sera de tester toutes les 4.294.967.296 possibilités de valeurs **long** (entier signé sur 32 bits) . En voici un exemple de tel algorithme :

```
long   n = LONG_MIN ;

 while (1)
  {
       if ( Blackbox(n) == 32 L )
         {
            cout << n<< "generates 32 ! " ;
            break ;
         }
       if ( n = LONG_MAX )
         {
            cout << "Didn't find 32 !"
            break
         }
        ++n ;
  } .
```

Le problème avec l'approche de la force brutale ce n'est pas le fait qu'elle ne va pas trouver la réponse, car on peut supposer qu'on l'a trouvé, par cet algorithme, et donc on connaît l'input qui a généré l'output 32 . Donc le problème ne sera pas la réponse, mais plutôt le temps qu'il a fallu pour la trouver . La précédente boucle peut trouver la réponse soit après quelques passages, soit après plusieurs milliard de passages - d'exécutions de cette boucle-. Clairement, on a besoin d'une solution plus intelligente.

3.3. Configuration et implémentation

Nous allons construire un AG pour maximiser l'output de *Blackbox*, et faire des analyses concernant les influences des paramètres de l'AG sur la vitesse de convergence.

- **Populations**

La première tâche est de définir notre ensemble solutions, ou la **population** . Vu que l'input de Blackbox est un **long**, notre population sera constituée d'un vecteur - array - de longs qui seront testés par la fonction . Notre chromosome sera alors une chaîne de 32 bits. Nous cherchons la plus grande valeur retournée par Blackbox pour un certain membre - individu - de notre population; par conséquent, nous allons définir un deuxième vecteur de **longs** qu'il a la même dimension que le vecteur de la population, et ainsi ils les longs- peuvent contenir les valeurs correspondantes de l'output de Blackbox . En d'autres termes, l'élément 5 du vecteur de l'adéquation contient la valeur renvoyée - retournée- par *Blackbox* quand on lui fournit la valeur en input qui est stockée dans le cinquième élément du vecteur de la population .

Supposons qu'on n'a pas trouvé l'input - la valeur de l'entier- qui produit la sortie 32 , alors on a besoin de produire une nouvelle population formée à partir de l'ancienne, en utilisant les valeurs d'adéquation pour déterminer le succès - les chances- reproductif . La façon la plus simple pour accomplir cela est d'utiliser " la roue de la fortune" ou la roue de loterie pour sélectionner les individus .

- **L'évaluation**

En fait, l'idée de la fonction est , si on peut le dire, simple . On choisit un nombre aléatoirement (ici on a choisit la vitesse de la lumière) qui est situé entre le min. (ici -2.147.483.599) et le max. (+2.147.843.600) , et comme une seule valeurs de cet intervalle correspond à l'output maximum , il suffit de changer le nombre pour changer la fonction, tout en restant entre le min et le max . car ce qu'il fait notre fonction c'est de tester les nombres aléatoires qui étaient générés, et qui représentés une population, en les comparant à la valeur de test(en fait on compare leurs bits 1 à 1) , et il y a un seul parmi ces valeurs d'input (de -2.147.483.599 à 2.147.483.600 , car ceux-là représentent un entier signé sur 32 bits , 2^{32}), que ses 32 bits correspondent aux bits de la valeur de test 299.792.458).
Ainsi pour les valeurs d'input, leurs fitness est évaluée en nombre de bits correspondants aux bits de la valeur de test (et de ce fait la fitness est entre 0 et 32). Donc il suffit de changer de valeur de test pour changer de fonction, et ainsi on simule parfaitement une fonction mathématique.

C'est pourquoi on ne propose pas la fonction comme paramètres d'entrée (pour plus de détails et d'explications concernant la fonction, voir Annexe,6.Configuration et implémentation, remarques sur la fonction d'évaluation et son code), car de toute façon l'AG est presque totalement (seulement la fonction d'évaluation, et elle seule, dépend de la fonction à optimiser) indépendant de la forme ou de l'expression analytique de cette fonction, et de la valeur de test, qui d'ailleurs, peut prendre 4.294.967.200 valeurs différentes.

- **La sélection (La roue de la fortune)**

Vu que nous l'avons déjà étudiée au § 2.3.3.,alors nous n'allons pas le décrire de nouveau .

- **Le crossover**

Ici nous avons utilisé le crossover par un poin que nous avons déjà vu au § 2.3.4.

- **La mutation**

Nous l'avons déjà vu, lui, aussi au § 2.3.3., mais il y a deux remarques à faire :

1) Un bon générateur de nombres aléatoires, qu'on l'utilise pour générer deux nombres aléatoires- qui représentent les probabilités- pour chaque chromosome afin de savoir si on va effectuer le crossover ou pas et la mutation ou pas sur ce chromosome, génére une surcharge substantielle de calcul (computational overhead), et cela pour toutes les chromosomes de la population d'une génération, et pour toutes les générations. Ainsi, par exemple, pour une population de 50 chromosomes *long*- chaque

chromosome est entier de 32 bits- on devrait utiliser 1600 nombres aléatoires durant la mutation, et ici on ne parle pas des 50 éventuels nombres alétoires pour effectuer ou pas les mutations, et autant pour les crossovers. Et cela, uniquement pour une génération.

2) Quand on veut exécuter une mutation, on utilise la probabilité *mrate*- ou P_m que l'utilisateur fournit comme un paramètre d'entrée-afin de savoir si un bit du chromosome sera aléatoirement changé ou pas. Cette mutation exige,donc, la génération d'un ou de deux nombres aléatoires - le premier pour décider si il y aura une mutation ou pas, et le second pour déterminer le bit - sa position- qui sera changé. Pour permettre de changer plusieurs bits dans un chromosome, on peut implémenter cela de la façon suivante :

```
While ( rand( ) < MUTE_CHANCE )
    mutate( chromosome ) ;
```

- **L'élaboration**

Le crossover et la mutation sont connus comme des **opérateurs de reproduction** . HOLLAND a aussi défini un autre opérateur de reproduction, **l'inversion** , qui change- inverse - un *segment* de bits dans un chromosome . Plusieurs chercheurs ont sérieusement étudié l'utilisation de l'inversion, et ils ont suggéré son utilisation seulement quand la taille du chromosome est " grande" . Ici on ne va pas l'utiliser.

D'ailleurs, même si on ne l'utilise pas, **il ne faut pas perdre de vue la taille du chromosome** quand on va implémenter le crossover et la mutation . Dans un chromosome de 4 bits seulement, 16 possibilités (4^2) existent; et le crossover ne va pas, probablement, créer aucun nouveau chromosome dans un tel environnement, car toutes les possibilités sont représentées avec plus ou moins la même probabilité vu que la population est petite. Par contre, la mutation , elle, a moins d'effet quand les chromosomes sont plus longs; changer un seul bit dans une chaîne de 256 bits a moins d'effet que changer un bit dans un chromosome de 16 bits.

Un autre facteur à garder en tête c'est la taille de la population ; plus grande est la population , plus de temps prendera le processus, mais des petites populations manquent - souffrent d'-une sélection robuste et efficace de chromosomes. Dans le design d'un AG on a besoin de faire l'équilibre entre la taille de la population et le nombre de générations- une génération est cycle complet de : évaluation - sélecticon-croisement- mutation- évaluation, requis pour trouver une solution . Par exemple, une population de 100 chromosomes peut trouver - on veut dire que l'algorithme peut trouver- la solution dans seulement 10 générations, mais chaque cycle d'évaluation peut prendre 4 fois plusque pour une population de 20 chromosomes qui , lui,peut trouver la solution en 20 générations. La population la plus petite- de 20 chromosomes- est 2 fois plus rapide que la plus grande, malgré le fait qu'elle trouve la solution en 20 générations au lieu de 10 et cela grâce - pour la petite- à la différence dans le temps d'évaluation(évaluer la fitness de 100 individus prend beaucoup plus que pour 20) .

La taille de la population a un autre effet sur l'AG : il dilue (diminue) l'influence des valeurs avec haute adéquation sur les succès reproductif . Dans une population de 10 chromosomes dans laquelle un chromosome a une fitness de 9, et les autres ont une fitness de 1, la moitié de tous les parents vont probablement, être sélectionnés parmi les 9 chromosomes qui sont relativement inaptes - unfit-, même si le meilleur chromosome est 9 fois plus adéquat.

Une autre règle empirique : l'évaluation des chromosomes et le calcul de la fitness sont - eux- qui consomment le plus de temps parmi les composants de l'AG. Notre but sera alors de réduire le nombre d'évaluations de fitness, soit en réduisant la taille de la population , ou en diminuant le nombre de générations requis pour trouver la solution.

Avant d'entrer dans le développement du code, on doit décrire deux techniques supplémentaires qui peuvent accroître les performances d'un AG. **La sélection élitiste** ou **l'élitisme** - qu'on a décrit au § 2.3.2.- copie toujours le plus adéquat chromosome d'une génération dans la génération suivante. utiliser l'élitisme garantit que la meilleure solution va survivre, et de ce fait s'assurer que les populations vont jamais céder du terrain- reculer- en fitness.

Un autre renforcement des performances, c'est la " **fitness scaling** " ou (la graduation de la fitness). En fait,c'est un changement d'échelle non-linéaire- ou la sélection par rangement, qu'on a vu au § 2.3.2.-. Pendant que la population converge vers une solution définitive, les différences entre les valeurs de fitness peuvent devenir très petites. Cela produit une roue de la fortune avec des sections presques égales, empêchant ainsi les meilleures solutions d'avoir un avantage significatif dans la sélection reproductive. La fitness scaling résout ce problème en ajustant les valeurs de fitness dans l'avantage des chromosomes les plus adéquats .

Le fenêtrage - windowing - est la forme la plus simple de cette sélection. Pour implémenter le fenêtrage, on commence par calculer comme d'habitude, les valeurs de fitness , en gardant une trace de la plus petite valeur de fitness. Puis on soustrait cette valeur minimum de toutes les valeurs de fitness, puis on ajuste le vecteur de fitness par rapport à la base zéro (en ajoutant 1, puis en élevant au carré le résultat de la soustraction; pour plus de détails cf. annexe 6. "l'implémentation"). On peut aussi soustraire une valeur légèrement plus petite que le minimum, afin de s'assurer que tous les chromosomes ont la même chance dans la reproduction.

- **L'implémentation**

Comme dans la discipline de la programmation classique la part de l'art est significative (le célèbre informaticien Knuth n'a-t-il pas appelé ses 3 volumes- de référence- : " The Art of Computer Programming"), en AG cette part est encore plus grande, et c'est surtout une question de " feeling" .

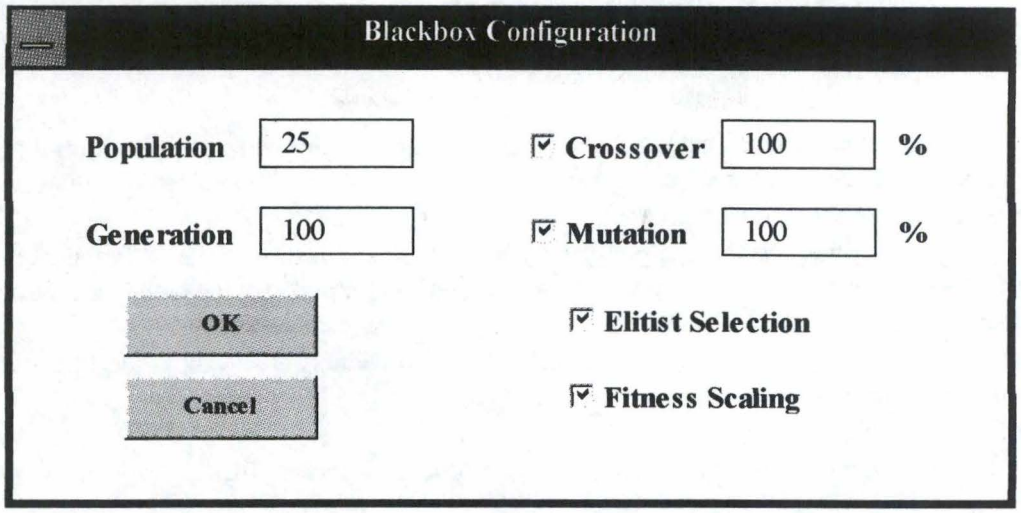
Pour comprendre exactement quels sont les facteurs qui produisent les plus rapides résultats dans une situation donnée, on a besoin de regarder comment ces facteurs agissent et qu'est ce qu'ils font.

Pour comprendre cela, on a défini un ensemble de paramètres pour notre problème *la boîte noire*-.On peut donner des valeurs différentes à ces paramètres, et ainsi on apprend et on comprend comment différentes combinaisons d'options affectent et influencent les performances.

On ne va pas aller trop loin dans les détails du code, mais on va décrire succintement les différentes fonctions, structures et étapes du programme, puis on va faire un résumé récapitulatif de l'algorithme (plus exactement de son implémentation) . Commençons d'abord par les paramètres .

- **Les paramètres agissant sur les checkboxes**

Pour définir, et manipuler, les paramètres on a dessiné une boîte de dialogue - windows dialog box- . Via cette boîte, on peut définir la taille de la population de l'ensemble des solutions- l'espace de solutions -, le nombre de générations à générer(à exécuter) .On peut aussi choisir, en activant ou pas- turn on and off - le crossover, la mutation, l'élitisme, la graduation(fitness scaling, ou le changement d'échelle), et en agissant sur les checkboxes, les valeurs des pourcentages permettent de déterminer les probabilités du crossover et de la mutation- les fameuses $\mathbf{P_c}$ et $\mathbf{P_m}$ - .La figure suivante illustre cette boîte de dialogue, avec les valeurs par défaut du démarrage du programme.



- **Le code**

On commence par créer une classe, BBOptConfig , pour manipuler les valeurs de l'input via la boîte de dialogue. L'objet BBOptConfig va contenir les paramètres de l'AG qu'on peut les retrouver via les différentes fonctions d'interrogation. Les valeurs statiques DefPopsize,Def...,Def..., contiennent le dernier ensemble des valeurs choisies des paramètres. Ces paramètres recoivent un ensemble de valeurs par défaut au démarrage- lancement- du programme.

Quand un objet BBOptConfig est instancié, il crée une boîte de dialogue remplie avec les valeurs Def... . Dans la partie suivante du code on implémente la boîte, et on vérifie si il y a des erreurs dans les valeurs de l'input de type : nombre de générations <= 0 , taille de la population < 0 ou < 10(ca n'a pas de sens, car on va avoir un chromosome de 3 bits), ...,etc. Une fois le constructor - de l'objet- a fini son travail, l'objet BBOptConfig agit comme un ensemble des valeurs constantes qui définissent les limites de l'AG . BBOptConfig est un exemple d'une classe qui encapsule une procédure spécifique à un environnement. N'importe quelle procédure qui veut utiliser l'objet BBOptConfig peut (et en fait, doit) ignorer la façon avec laquelle on assigne des valeurs aux paramètres (les paramètres sont déclarés **private**) .Si on déplace BBOptConfig vers un autre environnement, alors on doit changer le constructor - le constucteur, en OO- de la classe pour obtenir les valeurs utilisées par les techniques natives.

- **L'optimisation de la fonction (*Blackbox Optimization*)**

Ayant les paramètres en main, on peut maintenant écrire un AG pour optimiser l'output de la fonction *Blackbox*, et pour effectuer des statistiques afin d'étudier les influences des paramètres de l'AG sur la vitesse de convergence.

L'algorithme commence par créer l'objet BBOptConfig - comme on l'a déjà vu - pour définir et déterminer les paramètres de l'exécution. Puis on charge les paramètres dans des variables locales, pour éviter d'appeler des fonctions à l'intérieur des boucles imbriquées dans le programme (pas des appels croisés et dans tous les sens, mais plutôt une programmation clean et modulaire).
Après on alloue des buffers pour contenir la population, ses enfants ou descedants -produits par crossover et mutaion -, et les valeurs des fitness. Puis, on lance - sème- le générateur standard de nombres aléatoires de C++ (**srand( )**), et on définit les variables. La dernière étape de l'Initialisation est la création d'une population initiale avec des valeurs aléatoires.

La boucle principale se déroule de la façon suivante :

⇒ On commence avec la génération zéro (g = 0)
⇒ On boucle indéfiniment (**while(1)**), et on peut casser cette boucle (**break**) dans plusieurs endroits

⇒ On initialise des variables pour l'évaluation de la fitnesss, donc on initialise bestf, totf, minf (respectivement l'idividu de meilleur fitness,le total de fitness de tous les individus, et le minimum qu'on va l'utiliser en cas où l'élitisme était activé= **on**)
⇒ On teste - évalue- la fitness de toute la population,un individu à la fois, en utilisant la fonction *Blackbox* (cf.l'évaluation,plus haut), qui pour une valeur donnée d'input, renvoie la valeur d'output correspondante..
⇒ On garde une trace de la meilleure fitness - bestf -, et de l'individu - pop[i]-qui a cette fitness, et implicitement son indice i
⇒ On garde trace du min de fitness - minf -
⇒ On vérifie qu'on a eu au moins une certaine valeur de fitness (if totf == 0L)
⇒ On calcul la fitness moyenne (avgf)
⇒ Additioner (et peut-être graduer -scale -) les valeurs de fitness
⇒ Si (scale) alors vérifier que le minimum est un (car la fonction de fitness Blackbox renvoie entre 1 et 32);recalculer le total de fitness pour refléter les valeurs graduées (scaled)
⇒ Afficher des stast.(**statistiques**) concernant cette génération; Si c'était la dernière génération alors **break** (casser la boucle principale)
⇒ Créer nouvelle population par une boucle sur la taille de la population de la façon suivante :
1) Sélectionner un parent par la roue (sel = (long) ((float(rand( )) / float(RAND_MAX)) * float(totf)) ;
2) Exécuter la reproduction par le crossover, si crossover est **on**, et (rand( ) / RAND_MAX) < P_c alors sélectionner deuxième parent; utiliser masque (1...1...1) des bits qui doivent être copiés; construire une nouvelle chaîne à partir de deux parents; sinon un parent; pas de reproduction par crossover
⇒ Mutation : Si (mutate) est **on** et (rand( ) / RAND_MAX) < mrate alors sélectionner le bit à changer; changer le bit avec un masque
Remarque : dans ce programme, **crate** et **mrate** correspondent respectivement aux $\mathbf{P_c}$ et $\mathbf{P_m}$.- utilisés comme symboles de probabilités de crossover et de mutation le plus souvent dans la littérature spécilisée- .On divise rand( ) par RAND_MAX pour cadrer le résultat entre 0 et 1 comme une probabilité.
⇒ Si élitisme est **on**, alors remplacer le premier élément par le meilleur
⇒ remplacer l'ancienne population par une nouvelle
⇒ incrémenter le nombre de générations (++g).

- **La construction des statistiques pour l'analyse [*Blackbox Optimization(Analysis)*]**

Un AG est un processus stochastique qui montre des performances variables. Pour affiner l'analyse de l'algorithme d'optimisation de la boîte noire, on a créé une version spéciale du code précédent - en y ajoutant quelques simples fonctions- pour faire des statistiques, y compris la moyenne des performances (le nombre de générations, et le temps d'exécution de l'AG en tick pour 100 exécutions de l'algorithme .

Nous devons d'abord distinguer les statistiques- ou plus exactement les résultats sous forme des statistiques- de l'optimisation, de ceux de l'analyse.Dans l'implémentation de l'optimisation, les résultats sont des statistiques sous une forme de tableau (cf. Annexe,7. Résultats et analyses), qui contient les champs- ou les colonnes- suivants :
le numéro de génération, le meilleur chromosome (ou meilleure solution) trouvé, la fitness moyenne et le minimum de fitness, tous les trois concernant cette génération (dont le numéro était cité au début). Ainsi, on peut observer l'évolution de l'AG et sa covrgence continue (en fait, la fitness moyenne progresse d'une façon non-décroissante).

En ce qui concerne l'implémentation de l'analyse- statistique- du comportement de l'AG sous l'influence des paramètres suivants:

crate- =P_c-, *mrate-* =P_m-,*population-* popsize-,*generation-* Testsize ou nombre de générations-, *elitist selection-* elitisme ou pas-, *fitness scaling-* changement d'échelle ou pas-.Donc, nous avons ajouté à l'implémentation de *l'optimisation de la fonction*, le morceau du code suivant, et quelques déclarations supplémentaires (en fait, pour l'utilisateur cela est transparent, et se présente sous forme de deux options,optimisation sans ou avec analyses (*Analysis*), qu'il peut choisir dans le menu principal) :

```
// calculer le temps écoulé

elapsed = clock( ) - start;
sum += elapsed;
if (elapsed < mint ) mint = elapsed;
if (elapsed > maxt) maxt = elapsed;
if (g < ming ) ming = g;
if (g > maxg) maxg = g;
sumg += g;
```

Clock( ) est une fonction de C/C++ qui fournit le temps du processeur. *start* et *elapsed* sont des variables du temps du type *clock_t* (qu'on le trouve dans le fichier header "time.h"). Nous avons déjà- au début du programme- initialisé *start* :

```
start = clock( );
```

mint et *maxt* sont, eux aussi, des variables du type temps. *mint* est initialisée à l'entier non-signé maximum :

```
mint = UINT_MAX;
```

et maxt est initialisée à 0. *ming*, *maxg* sont des variables du type *size_t* (header "limits.h") et ils sont initialisées :

```
ming = UINT_MAX;
maxg = 0;
sumt = 0L;
sumg = 0L;
```

Les statistiques pour l'analyse remplacent, ici, les statistiques de l'optimisation (d'ailleurs c'est avec les déclarations adéquates, les seuls changements par rapport au code de l'optimisation), et ils- ceux de l'analyse- sont les suivants :

◊ La génération moyenne "*gen average*" : float(sumg) / float(ANL_SZ)

Où ANL_SZ est une constante de l'implémentation (= 100U) qui exprime la taille de l'analyse,ou autrement dit le nombre de génération sur lequel on calcule les moyennes. On peut bien-sûr changer cette constante à l'implémentation, mais l'utilisateur ne peut pas le faire. La génération moyenne fournit le nombre moyen- sur 100 exécutions- de générations avant d'arriver à la solution recherchée pour la configuration utilisée (ou qui est en cours).

"*gen minimum*" et "*gen maximum*" sont respectivement égaux aux *ming* et *maxg* qui expriment les nombres minium et maximum avant d'atteindre la solution recherchée.

◊ Le temps moyen d'exécution- le temps du processeur- "*tick average*" :

```
float(sumt) / float(ANL_SZ)
```

Comme pour la génération moyenne, le temps moyen est calculé sur 100 exécutions, et "*tick minimum*" est égal à *mint*, et "*tick maximum*" est égal à *maxt*.

Enfin le dernier changement concerne la condition d'arrêt de la boucle principale. Tandis que pour l'optimisation, la condition d'arrêt est d'atteindre le nombre de générations fixé à l'avance :

```
if ( g == GEN_SZ)
  break;
```

Pour le programme de l'analyse, cette condition est satisfaite si le nombre de générations est atteint ou si le meilleur chromosome est trouvé(bestf == 32L) ou le total de fitness de toute la poulation est égal à 0 :

```
// exit if this is final generation, or solution was found or total fitness = 0

if ((totf == 0L) || (bestf == 32L) || (g == GEN_SZ))
    break;
```

- **Résumé**

Chaque génération commence par l'évaluation de fitness de chaque chromsome en appelant Blackbox. La meilleure fitness- ou plus exactement l'individu avec cette meilleure- est tracée pour les besoins du rapport, tandis que la plus basse - la pire- est déterminée pour le cas où la fitness scaling est actionnée(**on**) . Dans le cas invraisemblable où la population a un total de fitness égal à zéro, la routine(le programme) s'arrête.

Après avoir calculé la fitness moyenne de la population et rapporter les statistiques de la génération courante, l'algorithme exécute la fitness scaling (à supposer qu'elle était sélectionnée) . Pour ajuster les valeurs de fitness, on soustrait le minimum des valeurs de fitness- qu'on a déjà sauvé- (technique de **windowing**), on ajoute 1, et puis on met le résultat au carré. L'addition de 1 donne à chaque chromosome une chance de reproduction, et mettre au carré la valeur de fitness " fenêtrée" (windowed) renforce les chances reproductives des meilleurs chromosomes.

Une troisième boucle implémente la reproduction. En utilisant la technique de la roue, l'algorithme sélectionne 1 ou 2 parents (cela dépend du fait si le crossover est **on** ou **off**).Dans cet algorithme on a utilisé un masque de bits à 1 , avec déplacement à gauche (one shifted bitmask[8] ; pour simplifier (111100) and (abcdef) ⇒ (abcd00)), et un autre bitmask pour la mutation d'un chromosome enfant . Si l'élitisme est activé, l'algorithme copie automatiquement le meilleur chromosome de la génération des parents au premier (dans l'ordre de fitness) élément de la nouvelle population . Puis on copie - on superpose - la nouvelle population au-dessus de l'ancienne - qui sera donc écrasée -, et on recommence à créer la génération suivante.

Une fois toutes les générations ont passé (cela dépend du nombre de générations qu'on a entré comme paramètre - *TestSize* -), la routine s'arrête en supprimant - deleting - les différents buffers :
delete [] pop ; delete [] newpop ; delete [] fit ;

Quand l'utilisateur choisit le programme "*Blackbox Optimization (Analysis)*", le même programme se déroule, mais, comme nous l'avons déjà vu dans le paragraphe précédent, des autres statistiques sont effectués, en calculant les mêmes résultats, mais pour 100 exécutions du programme précédent (en fait, le deuxième programme *Blackbox Optimization (Analysis)* est une version spéciale du premier programme *Blackbox Optimization*), donc en calculant les moyennes pour 100 exécutions. La condition d'arrêt du deuxième programme est différentedu premier: ici on s'arrête soit parce que le nombre de générations est atteint, soit parce que le total de la fitness est égal à 0, ou enfin- et c'est ici la nouveauté- le meilleur chromosome (ou la meilleure solution) est trouvé.

Il nous reste de décrire un peu plus précisément la fonction *Blackbox*- proprement dit, qui est en fait la fonction d'évaluation- et ainsi on a couvert tout le code .

---

[8] Le masque = 1..1..1(32 bits à 1) ; on a effectué les opération suivantes:
((float(rand( )) / float(RAND_MAX) ⇒ [0,1] ; (int)([0,1] * 32 ⇒ entier ε [0,32] le nombre de bits à shifter; on shift [0,32] bits du masque à 1, à gauche et on remplit à droite par des 0 ⇒ 11..10..0 ; et enfin l'instruction :
mask = oxFFFFFFFF << (int)((float(rand( )) / float(RAND_MAX) * 32.0F);

Commençons par décrire son code :

```
long  Blackbox ( long x )
{
   // valeur de test-- la vitesse de la lumière en mètre/seconde
   static const long n = ox11DE784AL; // la valeur en hexadécimal de 299.792.458;

long fit = 0L;
long mask = 1L;
 // on va copmter le nombre de bits qui correspondent - match -
       for ( int i = 0 ; i < 32 ; ++i )
           {
              if((x & mask ) == (n & mask ))
                    ++fit;
                  mask <<= 1;
            }
           // retourner la fitness entre 0 et 32
                   return fit;
  }
```

En fait, l'idée de la fonction est , si on peut le dire, simple . On choisit un nombre aléatoirement (ici on a choisit la vitesse de la lumière) qui situé entre le min. (ici -2.147.483.599) et le max. (2.147.843.600) , et comme une seule valeurs de cet intervalle correspond à l'output maximum , il suffit de changer le nombre pour changer la fonction, tout en restant entre le min et le max. car ce qu'il fait notre fonction c'est de tester les nombres aléatoires qui étaient générés, et qui représentés une population, en les comparant à la valeur de test (en fait on compare leurs bits 1 à 1) , et il y a un seul parmi ces valeurs d'input (de -2.147.483.599 à 2.147.483.600, car ceux-là représentent un entier signé sur 32 bits , 2^{32}), dont les 32 bits correspondent aux bits de la valeur de test 299.792.458.

Ainsi pour les valeurs d'input, leurs fitness est évaluée en nombre de bits correspondants aux bits de la valeur de test (et de ce fait la fitness est entre 0 et 32). Donc il suffit de changer de valeur de test pour changer de fonction, et ainsi on simule parfaitement une fonction mathématique.
C'est pourquoi on ne propose pas la fonction comme paramètres d'entrée, car de toute façon l'AG est totalement indépendant de la forme ou de l'expression analytique de cette fonction, et peu importe cette fonction - ou plus exactement la valeur de test qui ,d'ailleurs, peut prendre 4.294.967.200 valeurs différentes.

Ainsi, on a fini de décrire le code de ce programme . Son output, en utilisant les paramètres par défaut, est le suivant :

Pop. Size: 25
Test Size: 100
Crossover: true (100%)
Mutation: true (85%)
Scaling: true
Elitism: true

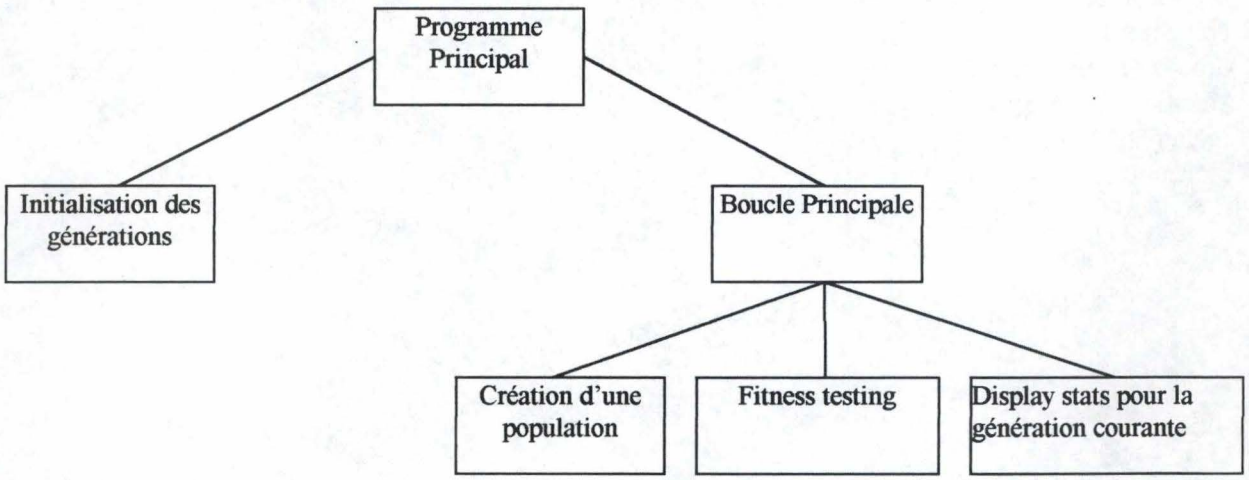
```
 0 best:       7b52 (20) avg. fit = 16 min. fit = 15
 1 best:       7a52 (21) avg. fit = 18 min. fit = 17
 2 best:    807a5a (23) avg. fit = 19 min. fit = 18
 3 best:       784a (24) avg. fit = 21 min. fit = 19
 4 best:       784a (24) avg. fit = 22 min. fit = 19
 5 best: 10847b4a (25) avg. fit = 22 min. fit = 21
 6 best: 10847b4a (25) avg. fit = 23 min. fit = 21
 7 best: 11807848 (26) avg. fit = 23 min. fit = 22
 8 best: 11907848 (27) avg. fit = 24 min. fit = 23
 9 best: 11907848 (27) avg. fit = 25 min. fit = 24
10 best: 11bef842 (28) avg. fit = 26 min. fit = 25
11 best: 11bef842 (28) avg. fit = 26 min. fit = 25
12 best: 10ce7848 (29) avg. fit = 26 min. fit = 25
13 best: 10ce7848 (29) avg. fit = 26 min. fit = 25
14 best: 10ce7848 (29) avg. fit = 27 min. fit = 26
15 best: 118e784a (30) avg. fit = 27 min. fit = 28
16 best: 118e784a (30) avg. fit = 28 min. fit = 26
17 best: 118e784a (30) avg. fit = 28 min. fit = 25
18 best: 118e784a (30) avg. fit = 28 min. fit = 27
19 best: 11de784a (32) avg. fit = 29 min. fit = 27
20 best: 11de784a (32) avg. fit = 29 min. fit = 27
```

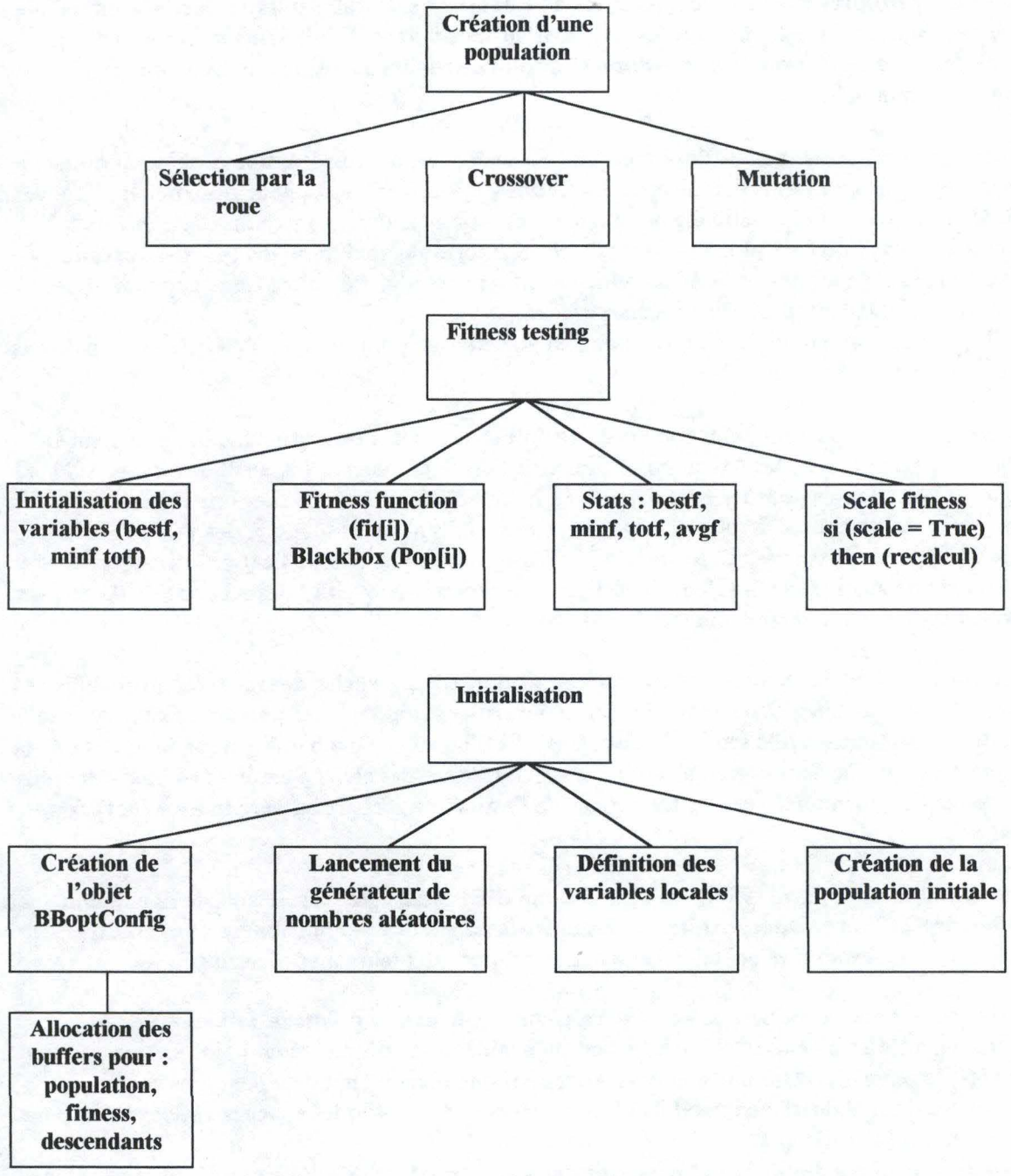
La première colonne est le numéro de génération, et la deuxième colonne est la valeur en hexadécimal du meilleur chromosome de cette génération, tandis que le nombre entre parenthèses est le nombre de bits matchés entre ce meilleur chromosome et la solution (le maximum) rechechée. *avg.fit* est la valeur moyenne- en nombre de bits matchés- de la fitness de tous les chromosomes de cette génération, et *min. fit* est la valeur minimum de fitness-toujours en nombre de bits- du pire chromosome dans cette même génération.

On remarque ici que le programme s'est arrêté après 20 générations, et parce qu'il a trouvé le maximum (**11de784a ≡ 299.792.458 la valeur de test**), malgré que ici il l'a trouvé (ce qui était voulu bien entendu), car on remarque bien que à la 19éme ou à la 18 ième génération, par exemple, le programme n'avait pas encore trouvé la meilleure solution, donc la condition d'arrêt ici est le nombre de générations.

Deuxième remarque c'est que la fitness minimum fluctue parfois -mais globalement elle avance -,mais par contre, **la fitness moyenne progresse continuellement, et d'une façon non-décroissante**.

Ainsi, on a fini de décrire le code de ce programme . Nous pouvons utiliser la technique HIPO (Hiearchy plus Input/Process/Output) pour représenter la structure top down du programme:





3.4. Conclusion Générale

En fait, cet exemple souligne et met en évidence les quatre différences entre les AG et les autres méthodes d'optimisation (ces quatre différences ou 4 caractéristiques,on les a déjà vu au § 4.5., mais ici on les voit sous une autre lumière, et dans le contexte particulier de ce problème). Donc,on va décrire brièvement ces 4 différences (pour plus d'explications cf [GOL 91]), pour en tirer des conclusions concernant, non seulement notre cas particulier (la boîte noire), mais aussi les autres cas en général.

1) Dans les autres méthodes d'optimisation, on pourrait travailler directement avec l'ensemble de paramètres (les valeurs de l'input dans notre cas), et changer leurs positions en utilisant les règles

de transitions propres à la méthode. Avec les AG, nous codons d'abord les valeurs d'input en une chaîne de longueur finie. Le fait que les AG utilisent les similarités de codage de façon très générale les rendent en grande partie libres des contraintes des autres méthodes (continuité, dérivabilité, unimodalité,etc) .

2) Dans beaucoup de méthodes d'optimisation, on se déplace avec précaution d'un point unique à un autre dans l'espace de recherche en utilisant une règle de transition pour déterminer le nouveau point. Cette méthode qui fonctionne point par point est dangeureuse parce qu'elle a une forte tendance à trouver de faux pics dans des espaces de recherche multimodaux (c'est-à-dire multi-pics) . En revanche les AG utilisent simultanément un ensemble de points (une population de chaînes), en escaladant plusieurs pics en parallèle.
En utilisant une population de diversité adaptée au lieu d'un point unique, l'AG " ne met pas tous ces oeufs dans le même panier" .

3) Souvent, les méthodes d'exploration nécessitent beaucoup d'information auxilaire pour bien fonctionner. En revanche les AG n'ont aucun besoin de cette information auxiliaire : Les AG sont aveugles (bien adaptés pour les problèmes type boîte noire) . Pour effectuer une recherche performante de structures de plus en plus intéressantes, ils n'ont besoin que des valeurs de la fonction à optimiser associées à chaque chaîne (ici le nombre de bits de l'entier-input correspondants aux bits de la valeur de test). Cette caractéristique fait des AG une méthode plus générale que beaucoup de procédures d'exploration.

4) Contrairement à beaucoup de méthodes, les AG utilisent des règles de transition probabilistes (P_c et P_m , la roue de loterie,...), afin de guider leur exploration. Pour les personnes habituées aux méthodes déterministes, cela semble bizarre, mais l'utilisation de probabilité ne signifie pas que la méthode n'est qu'une exploration aléatoire; les AG utilisent des choix aléatoires comme des outils pour guider l'exploration à travers les régions de l'espace de recherche, avec une amélioration probable.

Prises ensemble, ces quatre différeces - l'utilisation d'un codage, le travail sur une population, l'indifférence à l'information auxiliaire et les opérateurs alétoires - contribuent à la robustesse[9] des AG, ainsi qu'aux avantages qui en découlent par rapport aux tehniques plus classiques.

Bien sûr, ce sera naïf et injuste de notre part, si on ne cite pas les points de faiblesse des AG. D'abord, le fait d'être général et indépendant du problème est d'un côté un point fort; mais d'un autre côté, le refus d'utiliser une connaissance spécifique quand elle existe peut limiter la performance d'un algorithme quand il est comparé avec des méthodes conçues spécialement pour résoudre ce problème.
Il existe des manières d'utiliser l'information autre que la valeur de la fonction à optimiser, dans ce qu'on nomme les AG guidés par la connaissance (cf [GOL 91]).

Deuxièmement, en dépit de principes originaux et d'avantages incontestables, les AG souffrent d'une certaine inefficacité - pour les applications en temps réel - caractérisée par des temps de convergence généralement longs - pas pour les problèmes de décisions, et d'optimisation et d'ordonnancement où ils sont incomparablement meilleurs que les autres -, et un manque de précision lorsqu'une solution exacte est recherchée - car on approche beaucoup de l'optimale sans toujours, ou presque toujours, l'atteindre-.

---

[9] Pour rappel, pour Goldberg, la recherche sur les AG a pour souci principal l'amélioration de la robustesse, l'équilibre entre la performance et le coût nécessaire à la survie dans des environnements nombreux et différents.

Pour conclure, on peut dire que l'AG peut être rendu plus efficace si le on le combine avec des méthodes de recherche traditionnelles, habituellement plus " locales ". L'evolution (AG) et l'apprentissage individuel (méthode " grimpeur ") agissent de concert et s'aident mutuellement dans le processus global d'adaptation, et on a plus d'avantage, pour obtenir une efficacité optimale, à considérer des systèmes mixtes, car dans ce cas on marie leurs avantages respectifs et on réduit leurs incovénients .

Annexe

1. Quelques aspects de la complexité

Comme on l' a déjà vu dans le paragraphe précédent, les AG sont parmi les méthodes qui s'appliquent très bien aux problèmes difficiles à large espace de recherche et où les méthodes traditionnelles ne s'appliquent pas; beaucoup parmi ces problèmes ont été démontrés comme des problèmes NP-hard. Pour cette raison on va décrire succintemnt dans ce paragraphe les problèmes P, NP, NP-complets, NP-hard, et on va s'arrêter un peu plus sur le problème de satisfaisabilité (SAT) qu'on va le rencontrer de nouveau dans le chapitre 4.

Commencons d'abord par décrire la complexité. Intuitivement, un algorithme est efficace s'il n'utilise qu'une quantité "raisonnable" des ressouces nécessaires au calcul : le temps et la mémoire. Une telle définition peut sembler impossible à formaliser, et pour cela nous adopterons le principe que les ressources utilisées par un algorithme sont mesurées en termes d'une **fonction de complexité**. Plus précisement, les ressources utiliées par un algorithme sont caractérisées par la fonction exprimant la quantité de ressources en termes de la taille de l'instance du problème traité (à savoir du mot représentant l'instance du problème, où mot est à interpréter au sens de machine de Turing).

Nous ferons alors **l'hypothèse** que la frontière entre (fonction de) complexité **acceptable** et **inacceptable** se situe à la limite entre fonction **polynomiale** et **non polynomiale**. L'intérêt de cette hypothèse est triple. D'abord , elle rend la définition de la limite entre efficace et non efficace quasiment insensible au dispositif de calcul pour lequel la complexité est déterminée. D'autre part, elle rend aussi cette définition insensible à la représentation particulière utilisée pour le problème considéré. Finalement, elle est tout à fait significative en pratique. De plus, la pratique montre que lorsque un problème admet un algorithme de complexité polynomiale, il est très rare que cette complexité soit un polynôme de degré élevé (supérieur à 3 ou 4).

Dans ce qui suit, nous introduirons seulement quelques aspects de la complexité. Nous commencerons par définir la fonction de complexité en temps d'un algorithme en nous servant des machines de Turing comme modèle de référence. Ce choix peut paraître surprenant, mais il est parfaitement adéquat vu que nous nous intéressons uniquement à la distinction entre fonctions de complexité polynomiales et non polynomiales. Par complexité en temps des algorithmes on entend le temps nécessaire à leur exécution. Le temps d'exécution d'un algorithme donné dépend principalement de deux facteurs :
1- la machine utilisée (en pratique la machine, le système d'exploitation, le langage de programmation et le compilateur utilisés),
2- les données auxquelles l'algorithme est appliqué.

Plus précisément, on considère que le temps de calcul pour des données de taille n est le temps de calcul maximum por des données de cette taille. Cette convention porte le nom **d'analyse du cas le plus mauvais (worst-case analysis)**. On considère que complexité en temps donne une estimation du nombre d'opérations élémentaires nécessaires à l'exécution d'un algorithme, indépendamment de la vitesse d'exécution de ces opérations. En comparant les fonctions de complexité en temps d'algorithme pour des machines usuelles différentes, on constate que ces fonctions ne différent que d'un facteur constant, ce qui nous conduit à exprimer la complexité en temps d'un à un facteur constant près. Ceci a l'avantage de rendre cette mesure insesible à la représentation exacte des données : la fonction de complexité est idetique pour toutes les représentations dont la taile ne diffère que d'un facteur constant. Cette complexité est habituellement exprimée à l'aide de la notation O. Sa définition est la suivante :

- **Définition**

Une fonction g(n) est dite O(f(n)) s'il existe des constantes c et n_0 telles que pour tout $n > n_0$

$$g(n) = cf(n)$$

La notation O simplifie l'expression des fonctions de complexité. En effet, elle introduit implicitement le facteur constant nécessaire. De plus, elle n'exprime qu'une borne supérieur sur les fonctions et permet donc de donner une indication sur la complexité d'un algorithme sans nécessairement la caractériser complètement. Finalement, la notation O ignore le comportement des fonctions pour les petites valeurs, ce qui permet souvent de simplifier l'expression de la complexité d'un algorithme : pour des données de petite taille, la complexité d'un algorithme est souvent dominée par des opérations d'initialisation qui deviennent négligeables pour des données plus grandes.

Quand peut-on considérer qu'un algorithme est **efficace** ? Faut-il exiger une complexité en temps $O(n^3)$, $O(n^2)$ ou $O(n)$? Il n'y a pas de réponse absolue à cette question , mais quelques limites significatives peuvent être mises en évidence. Une **complexité exponentielle** ($O(c^n)$) ,où $c > 1$) est presque toujours excessive. Peut-on dire que toute fonction de complexité croissant moins vite qu'une exponentielle est acceptable ? En général, non. Toutefois, la différence entre croissance exponentielle et non exponentielle est suffisament forte pour qu'il soit tentant d'y fixer la limite de la complexité acceptable. Plus précisement, nous considérons comme **acceptable** toute fonction de complexité **polynomiale,** c'est-à-dire $O(n^k)$ pour un k fixé ! En d'autres termes nous identifierons les notions d'algorithme efficace et d'algorithme polynomial (de fonction de complexité polynomiale).

Après qu'on a défini la fonction de complexité polynomiale, il nous faut encore formaliser de façon précise les notions de **problème et d'algorithme polynomial .** Nous considérons uniquement des problèmes binaires (dont la réponse est "oui" ou "non"). Un problème sera représenté par un langage, à savoir **le langage de l'encodage** de ses instances positives[10] .

En toute généralité, un résultat de complexité polynomiale pour le langage encodant un problème sera aussi applicable au problème lui-même pour autant que cet encodage soit raionnable (cf. [Wol 91]), c'est-à-dire qu'il n'affecte la taille des instances du problème que de façon polynomiale.

Maintenant, on va définir le concept d'algorithme par celui de **machine de Turing**. C'est cette même notion qui va nous permettre de définir le concept d'algorithme polynomial . Ce choix est justifié par le fait que toute extension des machines de Turing et toute autre définition raisonnable de la définition d'algorithme polynomial coïncide avec celle obtenue à partir des machines de Turing.

- **Définition**

Soit M une machine de Turing déterministe qui s'arrête toujours. La **complexité en temps (time complexity)** de M est la fonction $T_{M(n)}$ définie par

$$T_{M(n)} = \max\{ m \mid \exists x \in \Sigma^* , |x| = n \text{ et l'exécution de M sur x comporte m étapes } \}.$$

La fonction $T_{M(n)}$ donne le nombre maximum d'étapes nécessaires à la machine de Turing pour décider un mot de longueur n. Il s'agit donc d'une complexité du "cas le plus mauvais" . Une machine de Turing sera dite polynomiale si sa fonction de complexité est bornée par un polynôme.

- **Défintion**

Une machine de Turing M est **polynomiale** (en temps) s'il existe un polynôme P(n) tel que la fonction de complexité $T_{M(n)}$ satisfait

$$T_{M(n)} \leq P(n) \qquad \text{pour tout } n \geq 0 .$$

La thèse içi est que , s'il existe un algorithme polynomial pour résoudre un problème , il existe une machine de Turing polynomiale qui décide le langage . Pour justifier cette thèse, il suffit de reprendre l'argumentation montrant que ce qui est calculable par une machine autre qu'une machine de Turing

[10] Les instances positives sont les mots représentant des instances du problème pour lesquelles la réponse est oui

l'est aussi par une machine de Turing. L'analyse des transformations utilisées pour établir cette propriété montre qu'elles sont toutes polynomiales !

- **La classe P**

La classe P (de **P**olynomial) est la classe des langages décidés apr une machine de Turing polynomiale. La classe P symbolise doc les langages recnnaissables par un algorithme efficace.Nous parlerons également souvent de problémes appartenant à la classe P. La classe P caractérise bien la notion d'algorithmes efficace : les algorithmes polymiaux sont souvent efficaces et les algorithmes non polymiaux ne sont presque jamais efficaces

La classe P a une définiton robuste: elle est insensible au choix particulier de machine et d'encodage .

Notons finalement que par analogie avec la notion de fonction calculable, il est possible de définier à partir des machines de Turing la notion de fonction calculable en temps polynomial.

- **Définition**

Une fonction est **calculable en temps polynomial** s'il existe une machine de Turing polunomiale qui la calcule.

- **Définition des transformations polynomiales**

Soient un langage L1 $\in \Sigma^*_1$ et un langage L2 $\in \Sigma_2^*$, **une transformation polynomiale** de L1 vers L2 (notation L1 $\propto$ L2) est une fonction f: $\Sigma^*_1 \rightarrow \Sigma_2^*$ qui satisfait les conditions suivantes :

1- elle est calculable en temps polynomial ,

2- f(x) $\in$ L2 si et seulement si x $\in$ L1 .

Les transformations polynomiales sont souvent appelées des réductions(fonctionelles) polynomiales . En termes de problèmes, une transformation est une fonction calculable en temps polynomial qui, à partir d'une instance x d'un problème P1, calcule une instance f(x) d'un problème P2 telle que l'instance x est positive si et seulement l'instance f(x) est positive. On peut décrire les transformations en termes de problèmes plutôt qu'en termes des langages encodant ces problèmes. Cela évitera la manipulation d'encodages particuliers.

- **Problèmes polynomialement équivalents**

Deux problèmes entre lesquels existent des transformations polynomiales peuvent être considérés comme équivalents du point de vue de l'existence d'algorithmes polynomiaux pour les résoudre. Cela se formalis comme suit :

- **Définition**

Deux langages L1 et L2 sont polynomialement équivalents (polynomially equivalent, notation L1 $\equiv_p$ L2) si et seulement si L1 $\propto$ L2 et L2 $\propto$ L1 .

La relation d'équivalence polynomiale entre langages est une relation d'équivalence (transitive, réflexive, symétrique). Elle définit donc des classes d'équivalence sur l'ensemble des langages. Chacune de ces classes regroupe des langages qui sont tous mutuellement polynomialement équivalents. Cela implique que, soit tous les membres de la classe ont solution polynomiale, soit aucun n'a de solution polynomiale. Par exemple, les problèmes HC[11] et TS (voyageur de commerce ou Travelling Salesman) font partie d'une telle classe d'équivalence.

La relation d'équivalence est très intéressante, car elle permet de remplacer un ensemble de questions concernant une classe de problèmes précis par une seule question concernant une clase de problèmes : la classe admet-elle oui ou non une solution polynomiale ,.

- **La classe NP**

Les algorithmes connus - on parle içi des méthodes classiques et pas des algorithmes génétiques, réseaux nueronaux ,...- pour résoudre un problème tel HC ne sont pas efficaces parce qu'ils ont pour principe d'essayer toutes les permutations possibles des sommets du graphe. Toutefois, pour chaque permutation, la vérification à faire (la permutation repésente-t-elle un circuit hamiltonien ?) peut être

[11] Le problème du **circuit hamiltonien(hamiltonan circuit**, abréviation **HC**). Le problème est de déterminer s'il existe un parcours fermé d'un graphe conteant chaque sommet du graphe une et une seule fois.

réalisée très rapidement. Nous sommes donc confrontés à un problème dont la solution algorithmique implique l'examen d'un grand nombre de cas, les vérifications à faire pour chaque cas pouvant être réalisées efficacement. Les autres problèmes polynomialement équivalents à HC ont la même caractéristique : c'est le nombre de cas à explorer qui rend leur solution algorithmique inefficace .

Pour exprimer que la complexité de la solution algorithmique d'un problème provient du nombre de cas à explorer , on peut dire que ces problèmes ont une solution efficace si l'énumération des cas ne coûte rien . Par exemple, HC a une solution efficace si l'on ne compte que le temps nécessaire à vérifier si **une** permutation des sommets représente un circuit hamiltonien . Cette vue est simplement une représentation du fait que la complexité de l'algorithme se trouve dans le nombre de cas à explorer plutôt que dans la procédure nécessaire au traitement de chaque cas . Son intérêt est qu'elle a une formalisation directe en termes de machines non déterministes .

Une machine non déterministe accepte un mot si elle a **une exécution** qui accepte ce mot. Imaginons alors que l'on fait correspondre chaque permutation à examiner pour résoudre une instance de HC à une exécution d'une machine ono déterministe , cette exécution acceptant uniquement si la permutation qu'elle définit est un circuit hamiltonien . Donc, la machine non déterministe accepte une instance de HC si et seulement si il existe une permutation définissant un circuit hamiltonien pour l'instance, à savoir , si et seulement si l'instance est positive. De plus, les exécutions de cette machine non déterministe sont coutes en fonction de la taille de l'instance. Nous allons donc maintenant étudier la classe des problèmes solubles efficacement (polynomialement) par une machine de Turing non déterministe .

- **Complexité des machines de Turing non déterministes**

Pour un mot d'entrée donnée, une machine de Turing non déterministe a des nombreuses exécutions .

- **Définition**

Le temps de calcul d'une une machine de Turing sur mot w est donné par

- la longueur de **la plus courte** exécution acceptant le mot si celui-ci est accepté,
- la valeur 1 si le mot n'est pas accepté .

- **Définition**

Soit M une machine de Turing non déterministe . La complexité en temps (time complexity) de M est la fonction $T_{M(n)}$ définie par

$$T_{M(n)} = \max\{ m \mid \exists x \in \Sigma^* , |x| = n \text{ et le temps de calcul de M sur x est } m \}.$$

- **Définition de la classe NP**

Une machine de Turing non déterministe est **polynomial** si sa fonction de complexité est bornée par polynôme .

- **Définition**

La classe NP (de **N**on déterministe **P**olynomial) est la classe des langages acceptés par une machine de Turing non déterministe polynomiale .

Des problèmes tels que HC et TS font partie de la classe NP . Les langages NP sont bien des langages décidables .

- **Théorème**

Soit L un langage appartenant à NP. Alors, il existe une machine de Turing déterministe M te un polynôme p(n) tel que M décide L et est de complexité en temps bornée par $2^{p(n)}$.

Ce théorème nous dit donc que tout langage appartenant à NP peut être décidé par un algorithme exponentiel . Peut-on faire mieux et par exemple montrer que tout langage appartenant à NP peut être décidé par un algorithme polynomial ? La réponse à cette question n'est pas connue à l'heure actuelle . Il y a toutefois de bonnes raisons de penser qu'elle négative. Pour élucider ces raisons, nous allons nous pencher sur la structure de la classe NP .

- **La structure de NP**

La notion de problèmes polynomialement équivalents permet de donner une structure à la classe NP . En effet, la relation $\equiv_p$ définit une partition de la classe NP dont chaque élément est un ensemble de problèmes polynomialement équivalents. De plus, il existe un ordre partiel sur ces classes d'équivalences .

- **Définition**

Une classe d'équivalence polynomiale C1 est inférieur à une classe d'équivalence polynomiale C2 (notation C1 $\underline{\pi}$ C2) s'il existe une transformation polynomiale de tout langage de C1 vers tout langage de C2 .

La signification intuitive de C1 $\underline{\pi}$ C2 est que , du point de vue de la complexité, la classe C1 est "moins difficile" que la classe C2. Parmi les classes d'équivalence polynomiale de NP, il y en a deux qui vont intéresser particulièrement : la plus petite (la plus facille) te la plus grande (la plus difficile) . La plus facile n'est autre que la classe P . Pour l'établir, nous devons montrer que P $\subseteq$ NP, que P est une classe d'équivalence polynomiale et finalement qu'elle plus facile .

- **Lemme**

La classe NP contient la classe P (P $\subseteq$ NP).

- **Lemme**

La classe P est une classe d'équivalence polynomiale.

- **Lemme**

Pour tout L1 $\in$ P et por tout L2 $\in$ NP on a L1 $\propto$ L2 ($\propto$: une transformation polynomiale)

- **Les problèmes NP-complets**

La classe des problèmes les plus difficiles dans NP est celle dite des problèmes NP-complets (NPC).

- **Définition**

Un langage L est NP-complet si

1. L $\in$ NP,

pour tout langage L' $\in$ NP , L' $\propto$ L.

Cette définition implique directement que la classe NPC des problèmes NP-complets est une classe d'équivalence polynomiale et que pour toute autre classe d'équivalence polynomiale C de NP on a C $\underline{\pi}$ NPC.

- **Théorème**

S'il existe un langage NP-complet L décidé par algorithme polynomial, alors tous les langages de NP sont décidables en temps polynomial, c'est-à-dire P= NP .

Les conséquences de ce théorème sont très intéressantes .Nous verrons que de nombreux problèmes tels que HC et TS sont NP-complets . Donc, on peut dire que ces problèmes n'ont de solution polynomiale que si P = NP . Autrement dit, pour démontrer qu'ils n'ont pas de solution polynomiale, il suffit d'établir que P $\neq$ NP !

Malheureusement, jusu'à présent, il n' a pas été démontré P $\neq$ NP . Cela signifie-t-il que les développements que nous venons de faire sont inutiles ? Pas vraiment . La classe des problèmes NP-complets comporte de très nombreux problèmes dont **aucun** n'a de solution connue ! Il est raisonnable de supposer que s'il existait une solution polynomiale pour un de ces problèmes, elle aurait été découvert.

Plus fondamentalement, il semble peu plausible que la capacité des machines non déterministes à explorer un nombre exponentiel de cas en temps polynomial n'apporte rien . Pour ces raisons, il est généralement admis que P $\neq$ NP . Notons qu'il a été demontré que si P $\neq$ NP , il existe des problèmes dans NP qui ne sont ni dans P ni dans NPC . Sous l'hypothèse que P $\neq$ NP, démontrer qu'un problème est NP-complet permet d'établir qu'il n'a pas de solution polynomiale . C'est l'argument que nous utiliserons pour établir que des problèmes tels HC et TS n'ont vraisemblablement pas de solution polynomiale.

Pour démontrer que, sous l'hypothèse P $\neq$ NP, il suffit se démontrer que, pour tout langage L' $\in$ NP , L' $\propto$ L sans nécessairement démontrer que L $\in$ NP . Dans ce cas, on dit que le langage L est NP-**dur** (NP-**hard**).

Quelle est la différence entre un problème NP-complet et un problème NP-dur , Tous deux n'ont vraisemblablement pas de solution polynomiale, mais le problème qui n'est que NP-dur porrait être

encore plus difficile que le problème NP-complet . En effet, le problème NP-complet est dans la classe NP, ce qui donne une borne supérieure à sa complexité : il y a des problèmes décidables qui ne sont pas solubles en temps polynomial, même sur une machine de Turing non déterministe !
En résumé , démontrer qu'un problème est NP-dur établit qu'il n'a vraisemblablement pas de solution polynomiale, démontrer qu'il est NP-complet montre aussi qu'il a au moins une solution polynomiale sur une machine de Turing non déterministe.

2. Quelques problèmes NP-complets

Nous commencons par le problème important SAT - qu'on va le revoir plus en détails au chapitre 4 - . Son importance vient du fait que une fois la NP-complétude de SAT est établie, il est possible de montrer par transformation que d'autres problèmes sont NP-complets . On peut dire que SAT = satisfaction des formules propositionnelles, ou plus exactement des formules en forme normale conjonctive . Pour simplifier içi - car on va revoir cela avec plus de détails - on va supposer qu'on a $W(A_1,....,An)$ une formule propositionnelle dont les variables sont $A_1,....,An$. Existe-t-il des valeurs logiques (**true**, **false**) pour les variables $A_1,....,An$ telles que $W(A_1,....,An)$ soit vraie ?
Pour formaliser un peu plus, on va dire que dans ces formules on utilisera des connecteurs logiques : $\neg$ $\wedge$, $\vee$, $\Rightarrow$,$\Leftrightarrow$,(,) et des varibles A_i dont chacune est représentée par l'entier i (décimal ou binaire)
Une formule $(A_1,...,A_n)$
est **satisfaisable** si il existe des valeurs logiques **(true,false)** pour les valeurs $A_1,....,A_n$ tel que $W(A_1,....,A_n)$ soit vraie.

Théorème de Cook :SAT est NP-complet.

On ne donnera pas ici la démonstration. Le lecteur intéressé peut la trouver dans [WOL 91].

On peut citer quelques problèmes NP-complets comme HC,TS -qu'on les a déjà vu -, le chemin le plus long entre deux sommets, couverture de sommets (VC), problème de la clique , Nombre chromatique d'un graphe, problème de partition, programmation entière, le problème Knapsack (le problème du sac à dos ou sac d'ordonnance), ...etc.

3. L'évolution biologique et l'évolution des espèces

Vres le milieu du 19éme siècle , Charles Darwin pensait que les espèces immuables vont devenir de plus en plus incompatible avec leur environnement qui varie tout le temps . La ressemblance de la descendance (la progéniture) à leurs parents lui a suggéré que les triats -les caractéristiques -passent d'une génération à la suivante ; il a aussi noté des légéres différences entre les enfants issus des mêmes parents , ce qui founit un réservoir d'individus différents , et qui se font concurrence pour la nourriture et l'accouplement .

De ces observations , Darwin a conclu que comme l'environnement changeait , les organismes les mieux adaptés aux nouvelles conditions vont donner des descendants portants les caractères réussis de leurs parents. Darwin a appelé ce processus **la sélection naturelle** qui était pour lui le mécanisme central de l'évolution des espèces.

La science moderne admet le fait que l'évlution est le mécanisme qui crée l'organisation biologique. Bien que la théorie de l'évolution a été raffinée à travers le temps depuis la mort de Charles Darwin, les concepts centraux -ou fondamentaux- de cette théorie sont restés intacts. On peut voir comment l'évolution se déroule aujoud'hui , et comment elle s'est déroulé dans le passé en examinant les fossiles

des espèces disparues ; et ainsi on peut voir comment les organismes se changent pour suvivre dans un monde en perpétuel mouvement.

Le minuscule papillon nocturne anglais founit un bon exemple classique de l'évolution en action . Avant la révolution industielle , les papillons de couleur claire se mélangeait aux lichens blancs des arbres , et ainsi ils se cachaient pour se protéger des oiseaux prédateurs ; Tandis que les mêmes papillons mais de couleur foncée étaient en contraste avec les lichens et souvent finir par être dévorés par les oiseaux . Mais quand la fumée du charbon utilisé dans les nouvelles usines de l'Angleterre a tué les lichens et a couvert les arbres de suie , les papillons de couleur claire sont devenus des cibles visibles et distinguées pour les oiseaux , tandis que ceux de couleur foncée se mêlaient avec l'environnement nouveau . En quelques années , la population de papillons était devenue , presque en sa totalité de couleur foncée , en s'adaptant ainsi à son nouvel environnement par l'action de la sélection naturelle.

Tandis que la survie des individus qui détermine les caractéristiques - ou les caractères - de la génération suivante c'est le succés de la reproduction de la population dans sa totalité qui détermine l'évolution d'une espèce.

La sélection naturelle est limitée par les caractéristiques d'une population ; Malgré qu'elle est souvent appelée la survivance - la survie - des mieux adaptés, la sélection naturelle opère parmi la survivance des meilleurs organismes disponibles . L'ajustement ou le "fitness" d'un organisme est relatif à plusieurs facteurs : l'écosystème changeant , les autres espèces , et les autres membres de sa population. Et ainsi ce qui est le "meilleur" ou le "mieux adapté" aujourd'hui (les papillons de couleur claire sur des lichens) peut ne pas être le "meilleur" demain (les papillons de couleur foncée sur la suie) .

Darwin ne savait pas comment les caractères passaient des parents vers leurs descandants, mais il a tout simplement constaté l'existence de ce passage . L'élément manquant était au-delà de la science de cette époque, et presque un siècle a dû passer après cette date avant que quelqu'un identifie l'agent mystérieux derrière cette évolution . En 1951, les biologistes Francis Crick et James Watson étaient les premières à décrire la molécule de l'Acide DésoxyriboNucléique (ADN) - une miniscule double hélice constituée , étonnamment , des éléments chimiques simples . L'ADN code les recettes des protéines et des enzymes de la vie , et il compacte une quantité énorme d'informations dans un espace extrêmement réduit ; si l'ADN d'une cellule humaine était déroulée, il meusureait à peu près 2 pieds(±61 cm).

Les biologistes sont toujours en train d'étudier et d'explorer cette piéce fondamentale du mystère de la vie . Chaque petit morceau de l'ADN contient des gènes qui définissent des parties déterminées du plan détaillé (blueprint)de l'organisme . Ces gènes sont les responsables du contrôle de chaque chose, de la couleur des yeux à la possibilité de développer certaines maladies .Le descendant hérite des caractères via les gènes reçus d'un parent . Les organismes simples comme les fungi (champignons) et les bactéries ont une reproduction asexuée qui se déroule en se duplicant . Une amibe unicellulaire , par exemple, crée des descendants en se divisant en deux organismes qui contiennent un identique ADN . Donc la reproduction asexuée produit des nouveaux organismes qui différent peu de leur géniteur , et se différent peu aussi entre eux .

La plupart des organismes complexes se reproduisent sexuellement en combinant les gènes des deux parents dans leurs descendants. En mélangeant et en appareillant l'ADN de deux organismes, la reproduction sexuelle augmente les variations et les différences dans une espèce. Les possibilités sont presque infinies ; par exemple, un couple humain peut produire plus que 7.000.000.000.000 plans détaillés (blueprint) différents d'une personne .

La collection d'informations génétiques dans une population constitue un réservoir génétique . Les grands réservoirs génétiques sont plus sains (en meilleur état) que les petits, car ils permettent de plus grand nombre de combinaisons génétiques. La grande variabilité signifie que un grand réservoir génétique est plus adaptable et être moins enclin aux désordres génétique récessifs. Un petit réservoir

génétique conduit à la consanguinité, ce qui augmentera la probabilité que les gènes récessifs vont se manifester dans la descendance des organismes très proches.

La sélection naturelle change les fréquences des gènes dans une population, mais elle ne produit pas des nouveaux gènes. Les premières formes de la vie sont apparues quand des molécules auto-réplicatives ont commencé à se rassembler dans une coopération mutuelle. Les premiers organismes complets ressemblaient à des amibes - et les amibes ne contiennent certainnement pas les génes nécessaires pour évoluer vers un être humain. Des nouveaux caractères ont dû apparaître d'une certaine façon, autrement les formes simples originales de la vie n'auraient jamais évolué vers les millions d'espèces qui se trouvent aujourd'hui sur terre.

Une mutation est un changement aléatoire dans les gènes d'un organisme. Il est hautement improbable qu'un changement génétique aléatoire va améliorer un organisme complexe qui est bien adapté à son environnement ; la plupart des mutations manifestes disparaissent par la sélection naturelle. Heuresement, la grande majorité de mutations n'ont pas d'effets . Les études de l'ADN humain ont montré qu'il y a des longues séquences de "gènes de rebut" qui servent apparement à rien ; Les mutations dans les gènes de rebut sont vraisemblablement dénuées de sens. Parfois les cellules peuvent réparer l'ADN endommagé, éliminant ainsi plusieurs mutations avant que celles-ci soient passées à des nouvelles cellules ou à la descendance .

La sélection naturelle mélange et tamise les réservoirs de gènes , en agissant sur les variations produites par la reproduction et la mutation . Parfois un réservoir génétique évolue d'une façon linéaire (en droite ligne) portant les espèces d'une espèce à l'autre , comme dans l'exemple cité plus haut de papillons nocturnes . Dans des autres cas, les forces de la nature agissent pour diviser le réservoir génétique, et la sélection naturelle agit sur les populations devenues séparées pour produire des nouvelles espèces. Si une espèce rencontre plusieurs niches ouvertes, elle peut rapidement se diversifier suivant un processus appelé la radiation adaptative (multiplication de différents éléments par mutation et sélection adaptatives). Quand les dinosaures ont disparu il y a 65millions d'années, ils ont laissé des niches inoccupées que les mammifféres ont vite occupé te exploité. Par la radiation adaptative, quelques petits mammiféres insectives comme la musaraigne sont devenus des miliers de types différents, allant des ours jusqu'à les êtres humains et les balaines .

Dans les années 80, les biologistes Niles Eldredge et Stehen Jay Gould ont introduit des modifications à la théorie de l'évolution . Ils ont postulé que l'évolution n'était pas un processus rigide constant , passant d'une espèce à une autre d'une façon continue . Leur examen des fossiles suggérait que les espèces restent statiques jusqu'à que des facteurs environnementaux forcent et déclenchent une évolution rapide vers de nouvelles formes . Cette idée connue sous le nom de **l'équilibre ponctuel** a été fortement dicuté et débattu , et elle parmi les idées les plus acceptées et les plus répandues .

4. L'organisation de l'information génétique

Qu'est-ce qu'un gène ? A quoi servent exactement les gènes ?
Imaginons, comme dans un film de science-fiction , un voyage vers l'intérieur d'une cllule d'homme ou autre organisme supérieur . Après avoir franchi la membrane qui la cerne, nous pénétrons dans le cytoplasme. Là, des foules de petites molécules subissent des transformations chimiques : là, s'élaborent les composants de l'architecture cellulaire , la production d'énergie chimique utilisable par la cellule. Tous ces phénomènes sont mis en oeuvre, catalusés par des protéines, les enzymes. Comment ces dernières sont-elles fabriquées ? par des petites usines moléculaires (les ribosomes) qui traduisent le mesage porté dans le cytoplasme par les ARN messagers (ARNm). Doù proviennent ces derniers ? Des gènes situés dans le noyau et dont ils sont les copies . Poursuivons notre voyage plus avant en remontant

le chemin des ARNm. Nous pénétrons dans le noyau ; il nous importe seulement de savoir qu'il héberge les chromosomes dont on sait , depuis de ce siècle, qu'ils sont le siège de **la mémoire génétique** .

Dans le noyau se trouvent 46 chromosomes . En vérité, 23 paires , car les chromosomes sont identiques deux à deux . Un jeu provient du spermatozoïde du père, l'autre de l'ovule de la mère, lors de leur fusion (les cellules sexuelles sont en effet les seuls à ne posséder qu'un seul jeu de chromosomes). Chacun de nos gènes est donc présent en double exemplaire, un sur chacun des membres d'un couple de chromosomes (exception faite des gènes présents sur les chromosomes sexuels). Toutefois, les deux copies d'un même gène peuvent être différentes, et leurs actions respectives se complètent ou, au contraire, s'affrontent. Si l'une des variantes (ou **allèle**) prend le dessus sur l'autre, elle est dite **dominante** tandis que l'autre est **récessive**. Nous possédons donc tous les mêmes gènes aux mêmes endroits, et c'est la multitude d'allèles qui fait que nous sommes, au bout du compte, tous différents.

Examinons un chromosome. Derrière cet aspect de bâtonnet boudiné se cache une molécule d'ADN, enroulée sur elle-même plus d'un million de fois. C'est l'ADN qui est **le support matériel de l'hérédité**- en d'autres termes, qui porte les gènes - tout comme la bande magnétique est, dans une cassette , le suport matériel de l'enregistement . Une cellule se divise un nombre limitée de fois, environ une centaine. Ensuite, elle meurt. A chaque division, la molécule d'ADN est également dupliquée.

Sur l'ADN, se trouvent les bases. **Quatre lettres suffisent à programmer toute la diversité du vivant** :A, G, T, C (adénine, guanine, thymine et cytosine). Un alphabet simple, et universl, dont l'enchaînement constitue un texte codant les gènes de l'individu. Chimiquement, ces bases, associées à des phosphates et des sucres, forment l'ADN. La molécule présente la particularité d'être double, faite de deux brins complémentaires qui lui donnent l'allure d'une immense échelle torsadée. Chaqun de ses barreaux est formé de l'appariement deux à deux des quatre bases, selon une règle immuable(ou presque, certaines mutations proviennent d'un mauvais appariement) : A s'associant avec T, et G avec C. Cette double nature de l'ADN est le mécanisme chimique même de l'hérédité. En effet, en séparant les deux brins- comme on ouvre une fermeture de chaque brin, la machinerie cellulaire parvient à obtenir deux nouvelles molécules identiques à la précédente.

Le **décodage** de l'information génétique se passe- sommairement- de la façon suivante:

1. Dans le noyau, l'ADN est **copié** en une molécule pratiquement identique, bien que simple brin, l'ARNm. Cette étape est appelée **transcription**.
2. Dans le cytoplasme, un **traducteur** de gène, le ribosome, s'arrime à l'ARNm, et **lit** les bases qui le constituent.

Grâce à la complexité d'autres ARN, dits de transfert(ARNt), les bases sont traduites, 3 par 3, en acides aminés correspondants. ATC est ainsi traduit en isoleucine, GGT en glycine, etc. Lassemblage des acides aminés forme la protéine. Ces deux étapes ensemble sont applées **traduction**.

Comme on l'a déjà dit, l'ADN est le support matériel de l'hérédité. Cet ADN est la pierre philosophale de la vie. Cette molécule contient et transmet une centaine de milliers de gènes, comme autant **d'instructions** pour formue alchimique extraordinaire, capable, à partir d'une cellule, de constituer un être vivant. Chaque gène occupe dans le chromosome -qui est une molécule d'ADN - un lieu précis, un **locus**. Par **mutation**, tout gène peut être transformé en un gène différent qui occupe le même locus ; ces gènes s'excluent mutuellement, ce sont des **allèles**.

Grâce aux gènes qu'elle renferme dans son ADN, la machinerie de cette première cellule va pouvoir fabriquer des molécules d'une variabilité extrême, tant dans leur rôle, que dans leur taille ou leur forme : les protéines. Une pour chaque gène. Celles-ci vont être à la fois les usines, les ouvrières et les matières premières permettant à la cellule de vivre, de se multiplier, de se différencier ensuite en cellules dédiées à un organe particulier et de s'assembler de manière à former un tout cohérent.

Ici, il y a quelques remarques importantes à faire :
D'abord, une très grande partie- jusqu'à 90%- de l'ADN des organismes supérieurs - ce que l'on appellera ici leur génome - est de type non-codant, c'est-à-dire qu'il ne code pour aucune protéine ou ARN .Cette première remarque est fondamentale puisque 90% de l'ADN humain est en apparence "inutile".
Ensuite, cet ADN génomique contient des quantités importantes de séquences répétées jusqu'à plusieurs centaines de milliers de fois, au point que les séquences non répétées, dites uniques, apparaissent plutôt l'exception que la régle. Pourquoi tout cet ADN non-codant ? Pouquoi tant de répétitions ?
Une explicatioin plausible est que l'existence de séquences répétées crée un énorme potentiel de variations génétiques brusques et importantes . Elle crée "de la mobilité" dans le génome responsable possible de "sauts" dans l'évolution moléculaire de l'ADN, et peut-être aussi, des espèces. Pour ce qui est des séquences non-codantes, la situation est beaucoup plus obscure. A l'évidence, si des contraintes structurales et fonctionnelles existent sur tout ou partie de l'ADN non-codant, elles sont moins fortes que celles qui s'exercent sur la portion codantes des gènes. Pour finir avec l'organisation de l'information gènètique, on va parler rapidement de la transmission, la recombinaison, le crossing-over et la mutation génétiques, car ces opérations interviennent dans les programmes évolutifs en général, et dans les algorithmes génétiques en particulier.

La transmission de l'information génétique d'une génération à l'autre se fait par l'intermédiaire de la **lignée germinale** ou germen, une filiation de cellules qui se différencient dès les premières divisions de l'embryon et finissant par former, chez l'adulte, les cellules reproductrices ou **gamètes** : ovules et spermatozïdes ou grain de pollen. Dans la règle, les cellules des animaux et des plantes sont diploïdes, c'est-à-dire qu'elles contiennent deux séries de chromosomes homologues et, par conséquent, deux exemplaires de chaque locus. Lors de la formation des gamètes, une division particulière a lieu, au cours de laquelle les chromosomes ne se dédoublent pas mais se séprarent , cest la **réduction chromatique**. Chaque gamète reçoit une série complète mais unique de chromosomes : les gamètes sont dits **halpoïdes**. Après la fécondation les noyaux des gamètes mâle et femelle fusionnent, ce qui rétablit le nombre diploïde de chromosomes.

Chaque individu porte donc, dans ses cellules deux séries de chromosomes dont chacune provient de l'un de ses parents. Si ses parents sont génétiquement différents, les gènes provenant de l'un et de l'autre sont en présence dans ses cellules, agissent ensemble mais ne se mélangent pas, ils gardent leur individualité. Au moment de la formation de ses propres gamètes, à la réduction chromatique, les deux séries parentales se dissocient : il y a **ségrégation** des gènes homologues. Les gamètes peuvent contenir l'un ou l'autre chromosome de chaque paire. Il en résulte une grande variété : ainsi chez l'homme, qui possède 23 paires de chromosomes, il peut se former, avec une propabilité égale, 2^{23} soit 8.388.608 sortes de gamètes différents, ce qui permet, par fécondation des gamètes tout aussi variés d'un autre individu, un nombre énorme de **combinaisons nouvelles** : chez l'Homme, $(8.388.608)^2$,soit plus de 70 billions.

De plus, les gènes qui sont situés sur un même chromosome ne passent pas toujours ensemble. Au cours des phénomènes qui précèdent la réduction chromatique, il arrive que deux chromosomes homologues(qui identiques dans leur forme,taille, le contenu et la distribution de leur gènes; malgré la présence éventuelle des différents allèles aux mêmes loci) se coupent au même endroit et échagent un de leurs tronçons; les gènes qui se trouvaient de part et d'autre de la coupure sont alors séparés et entraînés dans des gamètes différents, c'est le **crossing-over**. Il est d'autant plus fréquent entre duex loci que ceux-ci sont plus éloignés l'un de l'autre. D'autres événements réassocient entre eux deux fragments d'ADN issus de chromosomes différents. Tous sont appelés "recombinaisons". on sitingue classiquement la recombinaison dite "par homologie" qui intervient entre duex séquences d'ADN homologues (en pratique, le plus souvent, entre séquences alléliques, chacune portée par un des deux chromosomes homologues des cellulues eucaryotes, comme par exemple le crossing-over) et la recombinaison dite

"illégitime" . Celle-ci, plus rare, se produit par cassure et réunion de segments d'ADN n'ayant que peu ou pas d'homologie de séquences.

Ségrégations et recombinaison se font au hasard : la transmission des gènes à la descendance est affaire de **probabilités**, les résultats ne sont prévisibles que sur de grands nombres de cas semblables et s'expriment par des proportions statistiques.

D'autres événements peuvent altérer, au cours du temps, la structure de l'ADN . Les plus connus sont les **mutations ponctuelles** qui remplacent un nucléotide (le bloc de base de l'acide nucléique) par un autre ; ou selon une autre définition, la mutation est le changement survenu dans l'une des unités, ou gènes, qui sont les facteurs de l'hérédité. Ils peuvent concerner aussi plusieurs nucléotides, un ou plusieurs groupes de trois bases, et ils peuvent consister en suppressions ou en additions de tous les ordres de grandeur. Chaque mutation de chaque locus a une probabilité donnée de se produire et apparaît donc, dans l'ensemble, avec une fréquence moyenne qui correspond à la réalisation de cette probabilité : c'est ce qu'on nomme le **taux de mutation**, qu'on exprime en fonction du nombre de gamètes qui entrent en ligne de compte. Ce taux est toujours très faible : 10^{-10} à 10^{-4} selon les mutations(par exemple il est de 10^{-10} par nucléotide et par division cellulaire chez la bactérie E.Coli .

Dans la conception darwinienne, une mutation se répand dans une population si elle confère à l'individu qui la porte **un avantage sélectif**. Et pour finir, on définit l'allèle **dominant** comme celui qui masque l'expression d'un autre allèle du même gène, tandis que l'allèle **récessif** est celui dont l'expression est masquée par la présence d'un allèle dominant du même gène.

5. La génétique et l'évolution

Dans ce paragraphe on va jeter un coup d'oeil rapide sur la sélection et l'évolution des espèces, sans entrer pour autant dans les détails car cela dépasse le domaine de ce travail . La grande percée évolutionniste de Darwin, la sélection naturelle, reste le noyau dur de sa théorie. En accordant dans son système un rôle à "l'hérédité de l'acquis" , Darwin semble avoir diminué, implicitement, celui de la sélection naturelle qui fut repris avec vigueur par Weismann et par le néodarwinisme.

Des mutations- des transformations héréditaires brusques- furent observées dans le monde vivant depuis longtemps. Darwin, lui-même, a noté la race des moutons Ancona- fort courte en pattes-, grâce à ce qu'il appelait des "sports" ; ces grandes transformations soudaines apparaissaient à Darwin comme source d'accidents exceptionnels, voire de monstruosités, et non comme le matériel initial de l'évolution. L'ami de Darwin, Thoaas Huxley, regrettait que celui-ci n'ait pas pris en considération les grandes variations brusques qui, encore mieux que les variations graduelles- relativement insignifiantes-, auraient pu donner des matériaux à la sélection naturelle.

La théorie synthéthique de l'évolution- la plus plausible à l'heure actuelle- privilégie les micro-mutations comme matériel de base pour la sélection naturelle. Il est à noter qu'en disposant d'un énorme facteur temps l'on puisse finir par passer des mutations géniques à des changements chromosomiques. Mais il ne faut pas négliger non plus l'effet macroscopique remarquable des simples mutations géniques, donc des micro-mutations qui ont abouti, par exmple, à des nouvelles espèces de drosophile[12], pour ne pas parler d'autres êtres vivants. Ainsi, Philippe L'Héritier et ses collaborateurs de l'époque ont fait, vers 1936, des recherches sur une population de drosophiles pour vérifier l'explication donnée par Darwin selon lequel la sélection défavorise les insectes qui prennent leur vol vers le large, risquant ainsi, le plus souvent, de se noyer- comme le remarque le grand évolutionniste anglais- avec "l'avenir de leur race". En effet, les généticiens français observent que, sous l'effet du vent , l'élevage des drosophiles

[12] Mouche de vinaigre(Drosophila melanogaster), brun clair, longue de 2mm, qui offre un matériel de choix aux généticiens(nombreuses mutations, chromosomes de grande taille, cycle de reproduction très court).

situé sur le toit du laboratoire de Roscoff une sélection naturelle très dure qui ramène, en deux mois, les individus aptères- dépourvus d'ailes- de 12% à 65% .
Dans une contre-épreuve, quand l'élevage se trouvait à l'abri du vent, les drosophiles sans ailes disparaissaient en faveur des insectes ailés plus vigoureux par ailleurs. Ainsi les chercheurs français confirmèrent les observations de Darwin en prouvant la complexité d'action de la sélection naturelle qui ne favorise guère le"meilleur" ou le plus "apte" mais le plus adapté à un milieu donné : ainsi, l'aptérisme des insectes, paru par le hasrd des mutations, est maintenu par le jeu de la sélection étant une infirmité utile dans les habitats exposés au vent marin.

Le polymophisme génétique est illustré également par l'évolution des populations qui montre que la sélection n'agit guère seulement comme un agent conservateur du type normal de l'espèce, sauf, par exemple, certaines faunes cavernicole. Mais d'habitude on rencontre dans la nature une variation fluctuante des populations, une intégration des mutations nouvelles permettant un polymorphisme génétique indispensable aux évolutions futures. Jacques Ruffié motre que le polymorphisme génétiques des populations naturelles apparaît comme la réponse des groupes vivants aux variations incessantes du milieu et "ainsi, loin d'être défavorable, ce polymorphisme confère à ceux qui le portent un avantage sélectif puissant"[13].

En parlant du polymorphisme génétique et des mutations latentes existant toujours en chaque population, il s'agissait, bien sûr, de micro-mutations intraspécifiques. Peuvent-elles, comme la théorie synthétique l'envisage, rendre compte de grandes transformations biologiques comme, par exemple du passage des reptiles aux oiseaux? Le néodarwinisme classique suppose une série de micro-mutations orientées dans la même direction par la pression de la sélection naturelle. Pour autant, l'on ne peut pas exclure a priori ce que l'on peut appeler le "saltationisme" qui fait appel à des macro-mutations chromosomiques pour expliquer l'origine des espèces.

L'apparition de nouvelles espèces, la spéciation, ne peut être sans tenir compte de l'acquis de ce que l'on appelle "l'optique populationnelle". En effet, contrairement à la pensée typologique pour laquelle le type- voire le prototype ou l'archétype- d'une espèce est réel tandis que la variabilité n'est qu'une illusion- pensée inspirée à l'origine par Platon-, l'optique populationnelle considère que le type est une abstraction caractérisée par la moyenne d'un groupe, tandis que la variation est bien réelle. En fonction des rapports avec le milieu, on peut distinguer, par exemple, la spéciation sympatrique et allopatrique. La première formule implique une spéciation sans isolement géographique par rapport à l'espèce initiale, tandis que la seconde est une spéciationgéographique qui implique une spéciation de territoire. Dans les deux cas de figure, la population en évolution doit être séparée de la population initiale : dans le cas de la spéciation sympatrique, l'isolement est le produit d'une ou de mutations génétiques- dont nous avons déjà parlé-, dans celui de l'allopatrie, il s'agit de l'isolement écologique, géographique, comme celui existant, par exemple, dans les îles Galapagos où Darwin découvrit une sorte de laboratoire naturel de l'évolution des espèces.

Dans tous les cas de spéciation envisagés, on peut rencontrer les phénomènes d'anagenèse et cladogenèse. Dans l'anagenèse, considérée d'habitude comme une évolution progressive, ascendante, l'ensemble d'une espèce originelle devrait se transformer dans la nouvelle espèce fille; dans la cladogenèse, on rencontre une évolution en rameaux quand plusieurs populations d'une même espèce éloignées, en général, les unes des autres vers la périphérie de l'aréal occupé par l'espèce initiale donnent naissance à plusieurs espèces filles, adaptées chacune à son milieu, donc correspondant à une niche écologique particulière.

Que le milieu, opérant par l'intermédiaire de la sélection naturelle, mette une empreinte indélébile sur la structure des populations, plusieurs expériences irréfutable l'ont prouvé : ainsi celle, déjà citée, sur la sélection des mutations aptères chez drosophiles élevées en liberté sur la terrasse de la station de

[13] Jacques Ruffié, Traité du vivant, Fayard, 1982, P.156 .

Rosscof qui survivent tandis que les ailées, plus avantagées à d'autres ponts de vue, sont jetées à la mer par les vents locaux. Le milieu, pollué ou non, joue un rôle fondamental et certaines garnitures génétiques semblent, grâce à lui, évoluer sous nos yeux. Un exemple frappant- dont nous avons déjà parlé-, en ce sens, est celui qui regarde le noircissement de certaines espèces de papillons nocturnes dans les régions industrialisées d'Angleterre.

Ainsi la structure génétiques des populations se modifie grâce à la sélection- naturelle ou artificielle- produisant des êtres vivants plus adaptés aux conditions du milieu.

6. Implémentation

Remarques sur la fonction d'évaluation (*blackbox*)

Quelques mots d'explication à propos du programme s'imposent, car la définition est un peu ambigue ,et si nous utilisons un peu de mathématique cela peut clarifier et préciser les idées:

1) D'abord, cette technique de la "boîte noire" est souvent utilisée (cf. ch1[GOL91], §3.3.2.[REN95]) quand la connaissance du fonctionnement interne de cette boîte ne nous est pas nécessaire, et seulement l'entrée, la sortie et la relation entre eux qui nous intéresse vraiment; cela ne veut pas dire que ce fonctionnement est inconnu, mais ce qui nous intéresse réellement c'est la simulation du comportement d'un processus ou d'une fonction, indépendamment du fonctionnement interne de cette boîte.

2) Comme nous savons, pour définir une fonction f d'une variable réelle à valeurs réelles (dans notre cas tous les deux sont des entiers et donc aussi des réels), il faut se donner:
– une partie D de $\Re$ (ensemble des nombres réels) appelée domaine de définition de la fonction f;
– une règle qui à chaque élément x de D permet d'associer un nombre réel. Ce réel, noté le plus souvent f(x), est appelée image de x par la fonction f.

Donc, dans notre cas, D est l'ensemble des entiers signés codés sur 32 bits (-2.147.483.599 à +2.147.483.600), et l'image de f est aussi, ici, D l'ensemble des entiers signés sur 32 bits.
Cela est, en général, pour toute f dont la boîte simule le comportement, mais en ce qui concerne la fonction plus particulière dans notre cas, l'image est un sous-ensemble de D,et c'est [0,32].
Avec deux ensembles D, on peut définir un très grand nombre de fonctions, comme par exemple:
$x \rightarrow x$, $x \rightarrow 1/2x$, ou:
$f : [0,100] \rightarrow D$
$x \rightarrow x^2$
$]100,1000] \rightarrow D$
$x \rightarrow x^3$ etc.
L'importance est qu'on respecte la définition d'une fonction mathématique.

Dans notre cas, vu que l'application de l'AG est totalement indépendante- ce que nous allons montrer après- de la définition de cette fonction, et pour fixer les idées on a choisi une fonction f_c (**c** étant la vitesse de la lumière, 299.792.458 m/sec) qui est définie, par exemple comme suit:
$f_c : D \rightarrow D_c$ où $D_c = [0,32]$, tel que, par exemple, si $x = c$ alors $f(x) = 32$, sinon "peu importe", par exemple:
$x \in [0\text{-}100000] \Rightarrow f(x) = 0$
$x \in [(-1500)\text{-}(-1)] \Rightarrow f(x) = 1$
.
.
.

On aurait pu utiliser une autre fonction pour la même boîte noire, comme par exemple f_m (où m est 1 million) qui sera définie alors de la façon suivante:
$f_m : D \rightarrow D_m$, $D_m = [-100,100]$ tel que si $x = 10^6$ alors $f(x) = 100$, donc le maximum de f, sinon ..., ou la fonction f_5 qui définie de la façon suivante:
$f_5 : D \rightarrow D'$ où $D' = [minD,maxD-1]$
$x \rightarrow x/2$ si x est pair
$x \rightarrow x+1$ si x est impair

On peut remarquer qu'il y a, ici, trois fonctions:
La première,est une fonction en général $f : D \rightarrow D$, que la boîte noire peut simuler, et la deuxième qui est un cas particulier de f , et qui est dans notre cas $\mathbf{f_c}$ et la troisième est la Blackbox qui est la fonction d'évaluation ou de fitness-ou encore la mesure des gains-.L'implémentation, en C++, en FORTRAN ou en LISP, et l'efficacité de Blackbox dépend de $\mathbf{f_c}$,mais l'AG en soi (reproduction, crossover et mutation) n'en dépend pas. En fait, l'efficacité-en temps d'exécution ou en temps de convergence- dépend effectivement de l'implémentation de Blackbox, car c'est cette fonction qui évalue la fitness de chaque chromosome, sélectionné par l'étape de sélectionne, mais de point de vue algoritmique l'AG ne dépend nullement du code de Blackbox- qui est pour lui tout simplement une certaine fonction de fitness-, ni a fortiori de $\mathbf{f_c}$.

D'ailleurs, on touche ici aux deux points névralgiques- et dans un sens, deux points de faiblesse- des AG, qui sont le codage et la mesure des gains (ou de la fitness). Tous les auteurs et les chercheurs s'accordent sur le fait que l'efficacité et la réussite des AG dépendent fortement de ces deux points qu'on appelle parfois les mises en oeuvre des AG. Ces deux points dépendent, à leur tour, du problème et de l'environnement où l'AG sera appliqué et utilisé.

Dans notre cas, en ce qui concerne la mesure des gains on a choisi une certaine $\mathbf{f_c}$,ce qui enlève en rien à la généralité de la boîte noire, et à l'indépendance de l'AG- dans son fonctionnement- par à cette boîte, car l'implémentation de Blackbox sera très simple en C++, où on utilise un masque de bits à 1, et deux fonctions standard de C++: AND(&), et left shift(<< ; << oper1 oper2) qui shift- glisse- les bits de l'opérande$_1$ à gauche par un nombre spécifié par le second opérande, et remplit à partir de la droite avec des bits à 0. Le but de tout cela est de comparer deux chaînes de 32 bits, bit à bit, et puis compter le nombre de bits identiques- égaux- correspondants (qui ont la même position dans les deux chaînes). On voit que le nombre de bits identiques correspondants varie de 0 à 32, ce qui explique la raison pour laquelle on a choisi l'image de $\mathbf{f_c} = [0,32]$.

3) Le premier but de l'application Blackbox est que l'AG cherche la valeur $x \in D$ (l'input = 2^{32}) où $f(x)=32$(la valeur maximum de $\mathbf{f_c}$, donc en fait le but est maximiser $\mathbf{f_c}$ ou l'optimiser); bien sûr, dans notre cas cette x est égal à **c**(299.792.458), car nous avons voulu ainsi. En fait, cela se passe comme si nous avons soudé ou câblé l'intérieur de la boîte noire de cette façon afin qu'elle correspond- ou elle simule- la fonction $\mathbf{f_c}$, **mais** l'AG ne sait pas cela, et il est indépendant dans son fonctionnment de la forme de cette fonction. Le seul fait important est que cette fonction possède un maximum qui appartient à son image et qui correspond à une seule valeur appartenant à son domaine.

L'AG va chercher- intelligemment, et pas comme la force brutale- cette x qui maximise $\mathbf{f_c}$,et en ne testant au maximum que 500 valeurs possibles d'input(le nombre 500 est une valeur empirique tirée de l'expérience et qui dépend de notre cas-cf. annexe,7. résultats et analyses-) sur 4 milliards 300 millions valeurs possibles d'input, et cela grâce aux mécanismes suivants : évaluation, sélection, crossover et mutation.

Bien sûr, on aurait pu laisser l'utilisateur entrer ou choisir une autre valeur que **c**- on appelle dans le programme cette valeur *valeur de test*- à condition qu'elle se situe entre -2.147.483.599 et +2.147.483.600 (qui correspondent à l'input de la boîte noire qui est un entier signé de 32); en fait, cela

ne change rien en ce qui concerne l'AG, car supposons, pour fixer les idées, que l'utilisateur a choisi comme valeur de test 10^6 au lieu de (299.792.458), f_u au lieu de f_c . La forme de f_u peut être exactement la même que f_c ,à part que $f_u(10^6) = 32$ au lieu de $f_c(299.792.458) = 32$, et l'image de de f_u peut être- ou pas- exactement la même que f_c. La Blackbox reste la même dans les deux cas , car elle compare, toujours, les valeurs de l'input -une par une- avec la valeur de test, peu importe cette valeur, et elle- la Blackbox- compte le nombre de bits identiques correspondants entre ces deux valeurs, et quand ce nombre arrive à 32 elle s'arrête. Par contre, si on veut être tout à fait général, on peut laisser l'utilisateur changer, pas seulement la valeur de test, mais aussi la définition de f_c , et dans ce cas, si il choisit ,par exemple, f_5 , alors on aura une autre Blackbox, et par conséquent probablement, une autre fonction de fitness. Donc, les mécanismes de l'AG restent les mêmes, et ce qu'il change c'est -seulement- la fonction d'évaluation- ou fonction de gains- des chromosomes ou des individus, qui ,elle, dépend du problème, donc du codage et de la fonction de gains

Dans l'analyse et tuning(le deuxième but de l'application *Blackbox*), ce qui nous intéresse, ce n'est pas le fait que le programme- et donc l'AG- trouve la valeur de l'input x où f(x) = 32, mais en combien du temps(ou de générations) l'AG converge vers la valeur recherchée, et quels sont les paramètres- il y en a 6 en tout- qui influencent sur la vitesse de convergence et de quelles façons.

4) On a choisi que la sortie de la boîte noire soit 32 bits (ou 32 lignes) essentiellement pour deux raisons :

a) la boîte- en elle même- doit rester, relativement, générale, et si nous voulons changer(nous même et pas l'utilisateur) la fonction que la boîte simule, alors nous aurons beaucoup plus de choix si l'image de la fonction est D (pour la définition de D voir plus haut) au lieu d'un petit sous-ensemble(comme l'intervalle [0,32]) de D.

b) si la sortie était, par exemple, de 5 bits (ce qui est assez pour coder les valeurs de 0 à 31), et ainsi le maximum dans ce cas sera 31 (ou moins, si l'image de la fonction était un sous-ensemble de la sortie[0,31]), alors la tâche de trouver la valeur de x qui maximise la fonction -évidemment sans l'aide des AG- sera beaucoup plus facile, car on aura (2^{32} x 32) possibilités à la place de (2^{32} x 2^{32}) possibilités, ce qui est 2^{27} plus de possibilités dans le deuxième cas que dans le premier (cas de 5 bits). Cela compliquera substantiellement les investigations de quelqu'un qui cherche à trouver la valeur de l'input par des méthodes de type "force brutale", et montrera en même temps la force des AG.

7. Résultats et Analyses

Nous allons commencer maintenant par exécuter le programme avec divers paramètres, et essayer d'étudier les influences de ces paramètres sur la vitesse de convergence de l'AG en termes de nombre de générations, ou de temps d'exécutions en *ticks*, ce qui est pratiquement équivalent.Les paramètres que nous allons manipuler et varier sont six: **la taille de la population** qui doit être > 0 (en pratique comme nous allons le voir plus tard, cette taille doit être ≥ 10), **le nombre de générations** (> 0), **le taux de crossover** (l'option est désactivée ou **off ≡ 0%**, **on ≡ > 0%**), **le taux de mutation** (l'option est désactivée ou **off ≡ 0%**, **on ≡ > 0%**),**l'élitisme (off** ou **on)** et enfin **le changement d'échelle -fitness scaling-** qui lui aussi est (désactivée ou **off**,activée ou **on**).

Avec une population de 100 chromosomes, et **si le crossover et la mutation sont les seuls activés**, l'AG **n'a pas une bonne performance** :

Les paramètres :

Pop. Size: 100
Test Size: 500
Crossover: true (100%)
Mutation: true (100%)

Scaling: false
Elitism: false

Les résultats :

gen average = 497.43
gen minimum = 261
gen maximum = 500

tick average = 5606.19
tick minmum = 3010
tick maximum = 5767

Diminuer le taux de mutation, cependant, produit des résultats plus rapides en termes de nombres de générations, ou de ticks:

Les paramètres :

Pop. Size: 100
Test Size: 500
Crossover: true (100%)
Mutation: true (30%)
Scaling: false
Elitism: false

Les résultats :

gen average = 164.49
gen minimum = 45
gen maximum = 500

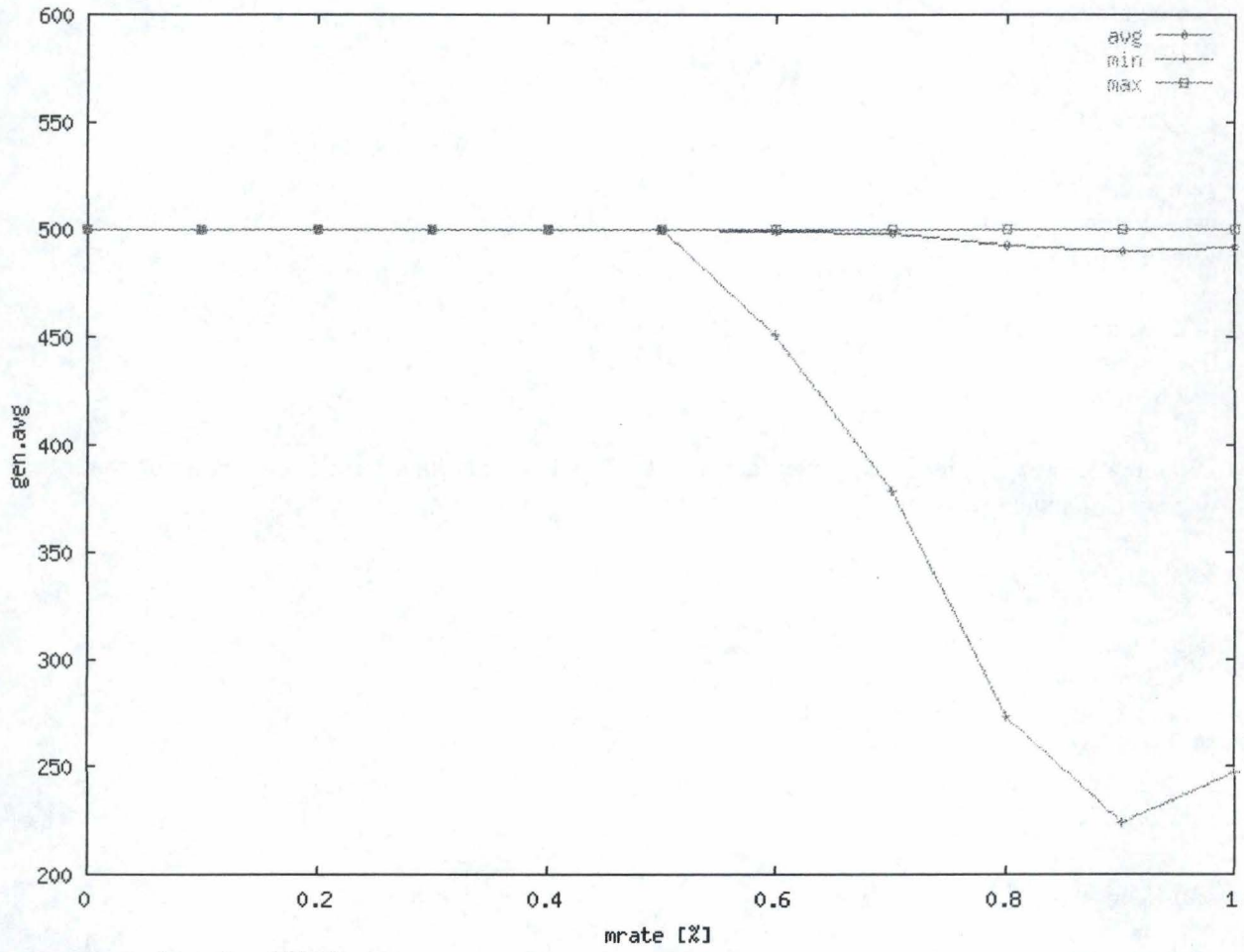
tick average = 1783.97
tick minmum = 494
tick maximum = 5493

Dans le tableau 1., et la figure 1.on voit les résultats des tests que nous avons effectués pour étudier la relation entre le taux de mutation (mrate) et la vitesse de convergence (mesurée en *gen. average*), en laissant constants les autres paramètres. Nous avons surtout fait attention sur les résultats quand *mrate* prend les valeurs recommendées -ou conseillées-dans la littérature spécialisée (0,5%- 1%), malgré le fait que ces valeurs sont empiriques, et d'ordre général, car comme nous l'avons cité à plusieurs reprises, en AG la mise au point de chaque AG est très important et constitue peut être (car la programmation est un art aussi; cf. plus haut § implémentation, et à la fin § conclusion générale) un de leurs points de faiblesse.

- **crossover (crate) = 100%**

mrate	0%	0.1%	0.2%	0.3%	0.4%	0.5%	0.6%	0.7%	0.8%	0.9%	1%
gen. averag	500	500	500	500	500	500	499.51	497.96	492.71	490.13	492.08
min	500	500	500	500	500	500	451	379	273	224	247
max	500	500	500	500	500	500	500	500	500	500	500

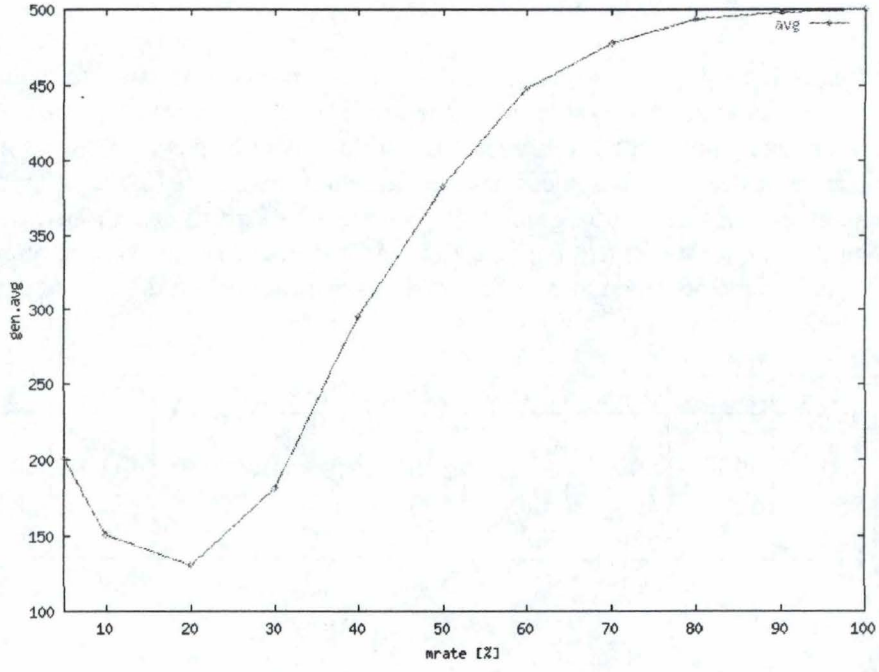
Tableau 1.



- **crossover (crate) = 100%**

mrate	5%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
gen. averag	200.72	150.86	130.33	181.84	295.06	383.21	448.32	478.22	493.73	498.22	500

Tableau 2.



Vu que le fait de diminuer la mutation- à lui seul- accélère l'algorithme, est-ce que la diminution du taux de crossover sera aussi avantageuse ? Non.Avec un taux de crossover mis à 90% , l'algorithme ralentit considérablement :

Les paramètres :
Pop. Size: 100
Test Size: 500
Crossover: true (90%)
Mutation: true (30%)
Scaling: false
Elitism: false

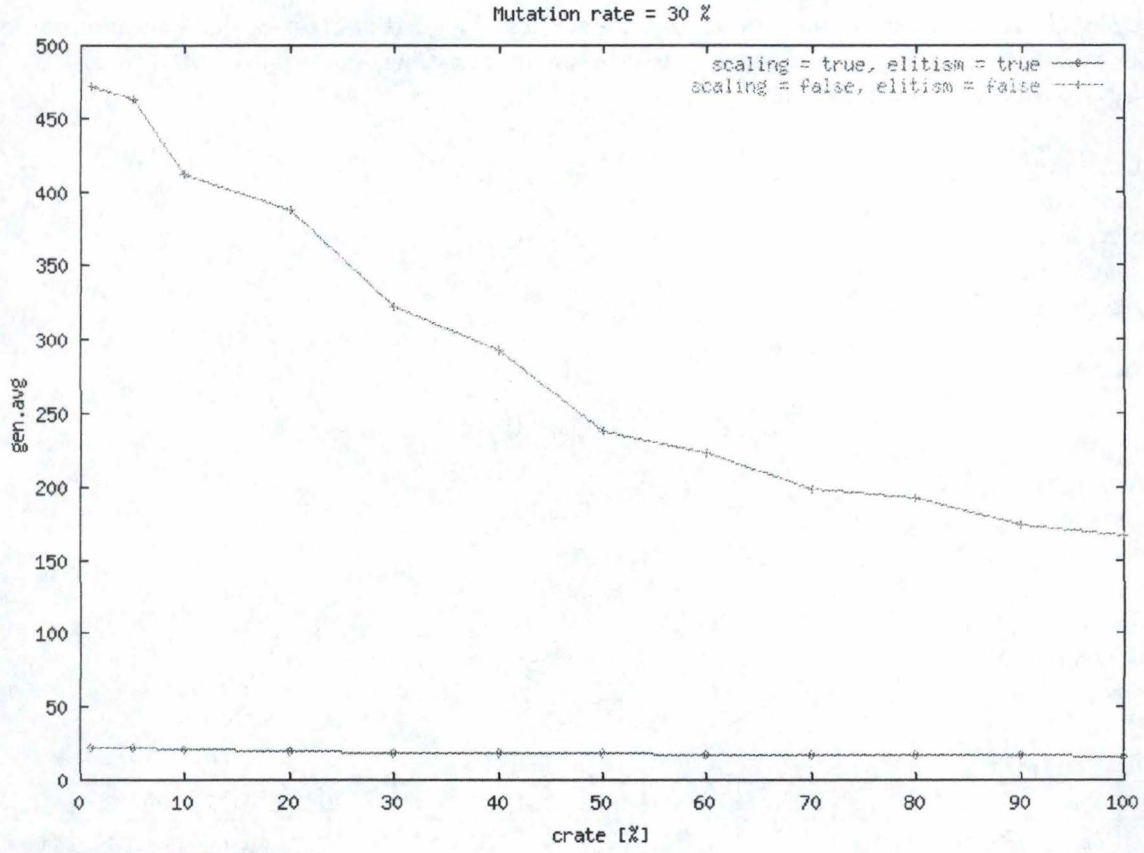
Les résultats :
gen average = 192.94
gen minimum = 66
gen maximum = 500

tick average = 2030.03
tick minmum = 696
tick maximum = 5273

- **mutation (mrate) = 30%**

crate	1%	5%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
gen. ave. Elit=T Sca=T	22.64	21.78	21.04	19.75	18.92	18.4	17.78	17.34	17.39	16.64	16.65	16.09
gen. ave. Elit=F Sca=F	472.3	465.3	412.3	387.0	322.3	292.5	238.1	223.5	198.1	191.9	174.5	166.1

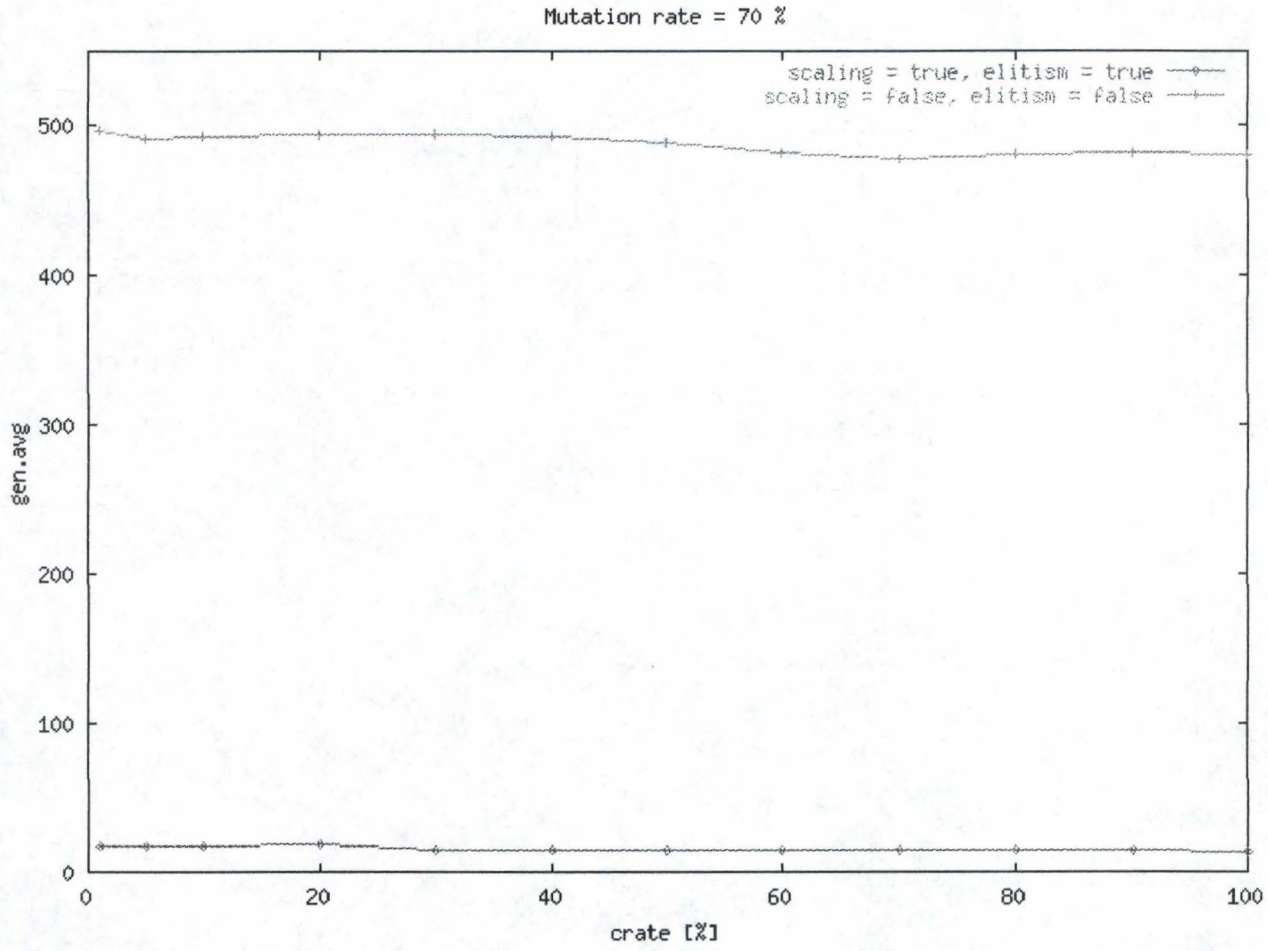
Tableau 3.1.



- **mutation (mrate) = 70%**

crate	1%	5%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
gen. ave. Elit=T Sca=T	17.22	16.99	16.90	18.80	15.50	14.99	15.09	14.67	14.63	14.18	14.34	13.70
gen. ave. Elit=F Sca=F	495.8	490.6	491.8	493.1	493.8	491.9	487.3	480.9	477.7	480.1	481.1	479.2

Tableau 3.2.



Dans le tableaux 3.1,3.2., et les figures 3.1.,3.2., on voit la relation entre le taux de crossover *crate* et la vitesse de convergence *gen. average* pour deux valeurs de *mrate* .

Notons aussi que si le taux de mutation était trop petit ou la mutation était désactivée - OFF -, l'algorithme tournera avec une qualité médiocre (cf.tableau 2., et figure 2).

Activer l'élitisme améliore la vitesse de l'AG, assurant le succès de trouver une solution avant que 400 générations soient déroulées :

Les paramètres :
Pop. Size: 100
Test Size: 500
Crossover: true (100%)
Mutation: true (100%)
Scaling: false
Elitism: true

Les résultats :
gen average = 130.34
gen minimum = 37
gen maximum = 392

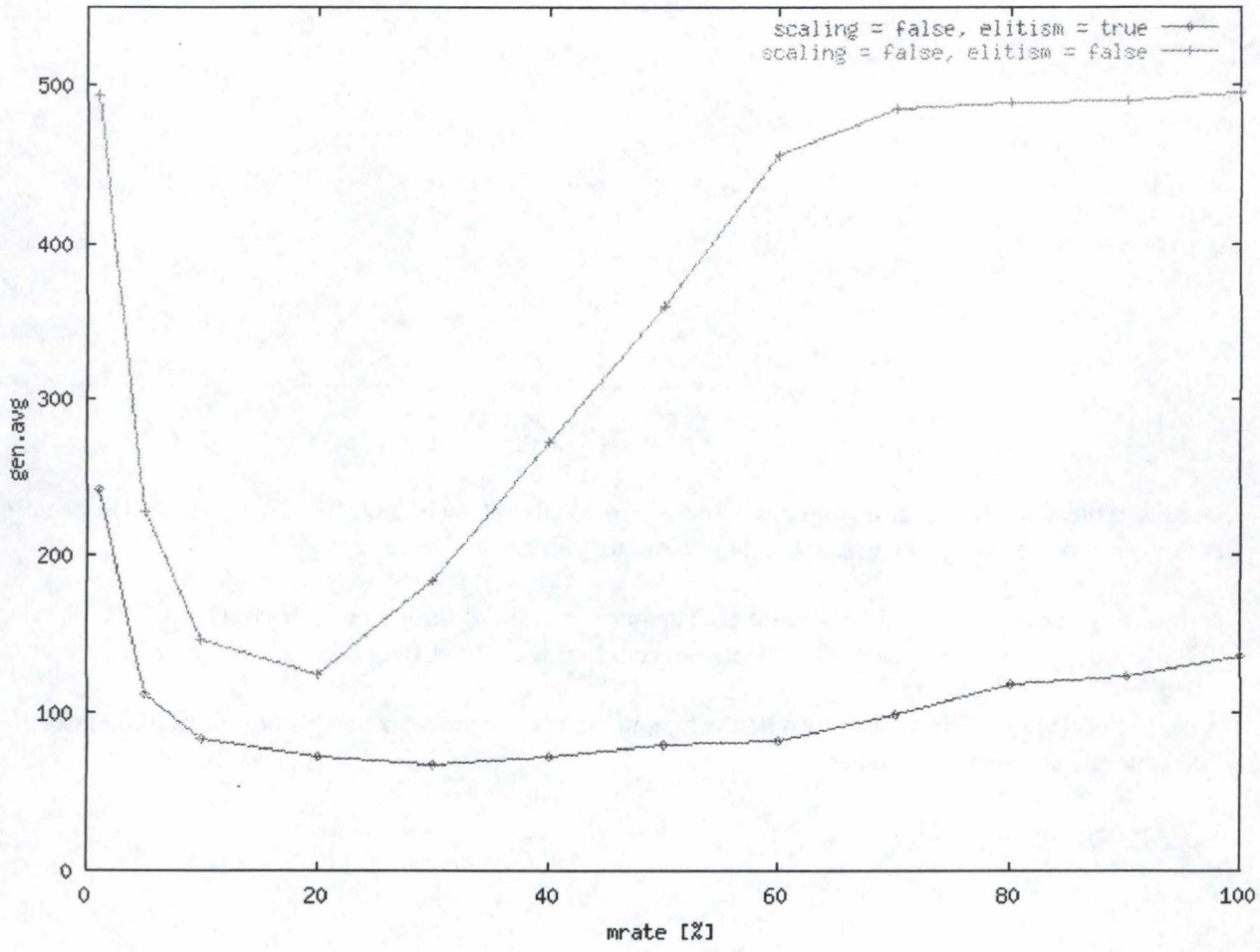
tick average = 1464.85
tick minmum = 415

tick maximum = 4394

- **crossover (crate) = 100%**

crate	1%	5%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
gen. ave. Elit=T Sca=F	242.2	112.1	83.96	72.28	67.83	72.74	80.57	82.80	98.80	118.4	123.9	135.7
gen. ave. Elit=F Sca=F	494.3	227.4	146.4	125.2	182.8	272.1	360.6	455.5	485.3	488.9	490.2	495.6

Tableau 4.



Nous remarquons dans le tableau 4. et la figure 4.qu'en activant l'élitisme, on améliore sensiblement (on diminue de presque de la moitié le nombre moyen de générations avant la convergence) les performances de l'AG.

Combiner un taux de mutation relativement bas (20%- 40%), avec l'élitisme activé, augmente encore la vitesse de l'AG (cf. taleau 4 et figure 4, et les résultats ci-après):

<u>Les paramètres</u> :
Pop. Size: 100
Test Size: 500
Crossover: true (100%)

Mutation: true (30%)
Scaling: false
Elitism: true

Les résultats :
gen average = 68.32
gen minimum = 32
gen maximum = 152

tick average = 742.59
tick minmum = 1
tick maximum = 1648

Toutefois, l'augmentation la plus significative des performances arrive quand la fitness scaling est utilisée en même temps que le crossover et la mutation (cf. tableau 5, figure 5):

Les paramètres :
Pop. Size: 100
Test Size: 500
Crossover: true (100%)
Mutation: true (70%)
Scaling: true
Elitism: false

Les résultats :
gen average = 14.12
gen minimum = 10
gen maximum = 18

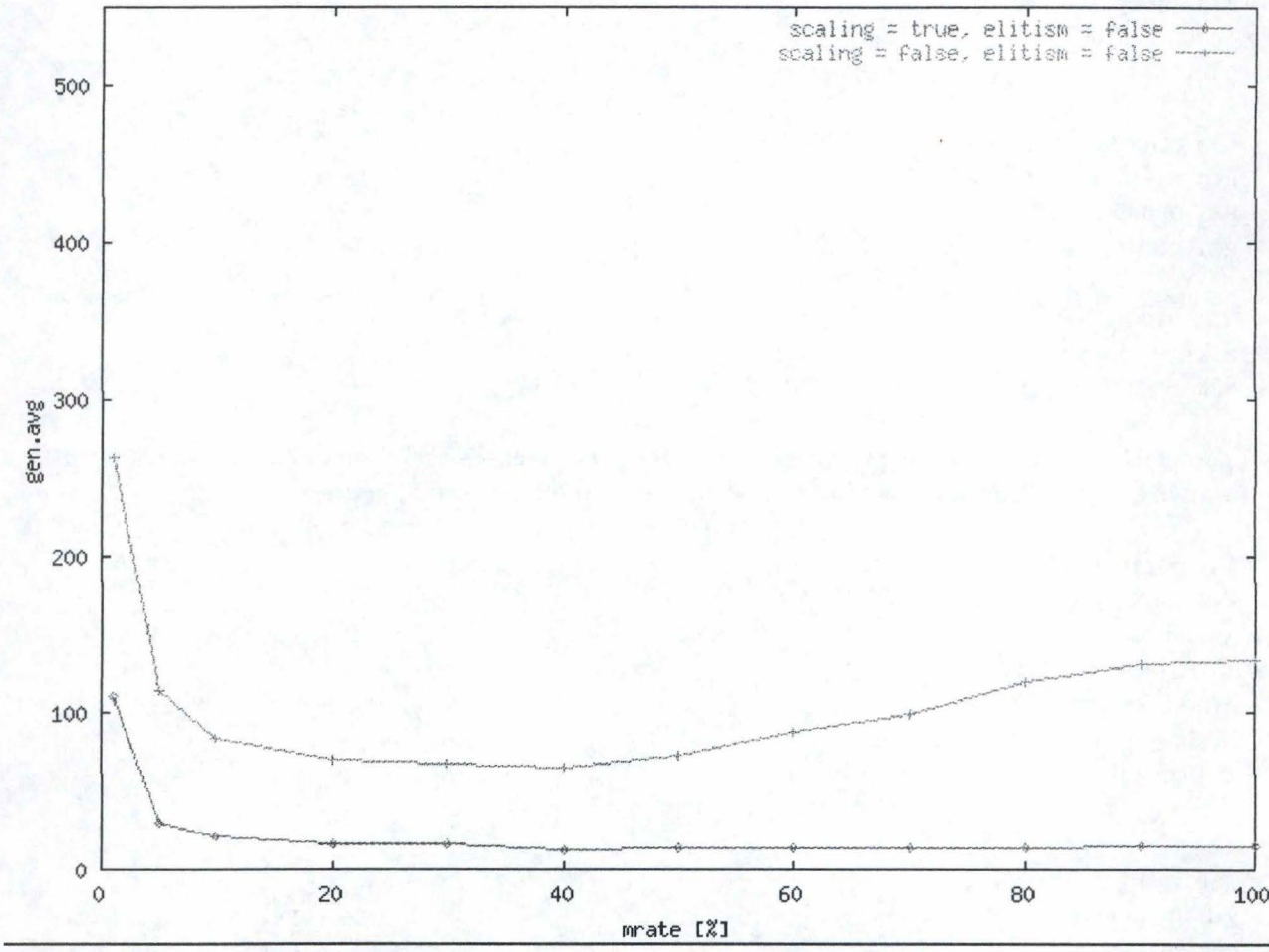
tick average = 161.48
tick minmum = 120
tick maximum = 220

Remarque:
parfois on a arrondi les chiffres, ce qui enlève en rien la précision des résultats, car les différences sont tout à fait insignifiantes par rapport aux chiffres et à leurs rapports relatifs.

- **crossover (crate) = 100%, elitism = False**

mrate	1%	5%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
gen. ave. Sca=T	111	30	22	17	16	13	14	14.1	14.2	14.6	14.8	14.9
gen. ave. Sca=F	264	115	84	70	68	65	73	88	99	120	131	134

Tableau 5.



On remarque la grande différence entre les résultats sans le scaling (Sca = F), et avec le scaling (Sca = T). On améliore de presque de 400% les performances de l'AG.
La raison de cette augmentation formidable de la vitesse de convergence est liée à la nature du problème qu'on est en train de résoudre. Les valeurs de fitness retournées par *Blackbox* sont situées dans un domaine relativement étroit, laissant ainsi peu d'écarts entre les différents chromosomes .Cela est connu comme étant le phénomène de la course serrée . L'inclusion de la fitness scaling ajuste le succès reproductif en faveur des chromosomes qui ont la haute fitness.

On doit aussi remarquer le changement du taux de mutation associé à l'arrivée de la fitness scaling.(cf. tableau 5, figure 5). Sans scaling, un taux bas de mutation est avantageux; cependant, une fois l'évaluation de la fitness a biaisé le succès reproductif (le biais de scaling), un taux de mutation plus haut est plus utile dans ce cas.

L'utilisation des quatre techniques, ensemble, produit un algorithme encore plus rapide (cf. tableau 6, figure 6):

Les paramètres :
Pop. Size: 100
Test Size: 500
Crossover: true (100%)
Mutation: true (70%)
Scaling: true
Elitism: true

Les résultats :
gen average = 13.61
gen minimum = 10
gen maximum = 19

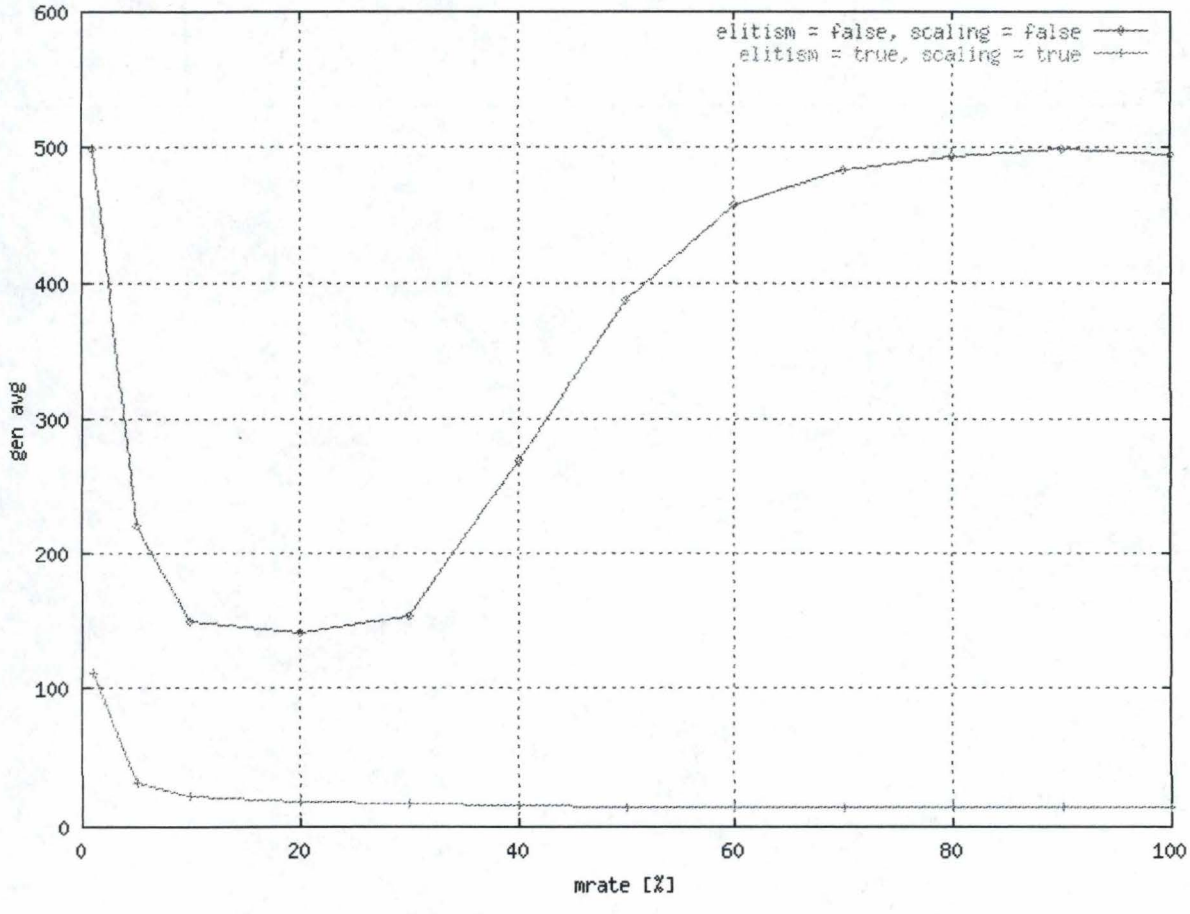
tick average = 155.44
tick minmum = 110
tick maximum = 220

Dans le tableau 6 et la figure 6, on a varié *mrate* en gardant *crate* constant(= 100%), tandis que dans les tableaux 7 et 8, on a varié *crate*, et ceci pour *mrate* = 30% et *mrate* = 70%.

- **crossover (crate) = 100%**

mrate	1%	5%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
gen. ave. Elit=F Sca=F	498	221	149	140	153	269	388	457	484	493	498	494
gen. ave. Elit=T Sca=T	111	32	22	18	16	15	14	14	**13.61**	13.73	13.96	14.34

Tableau 6.



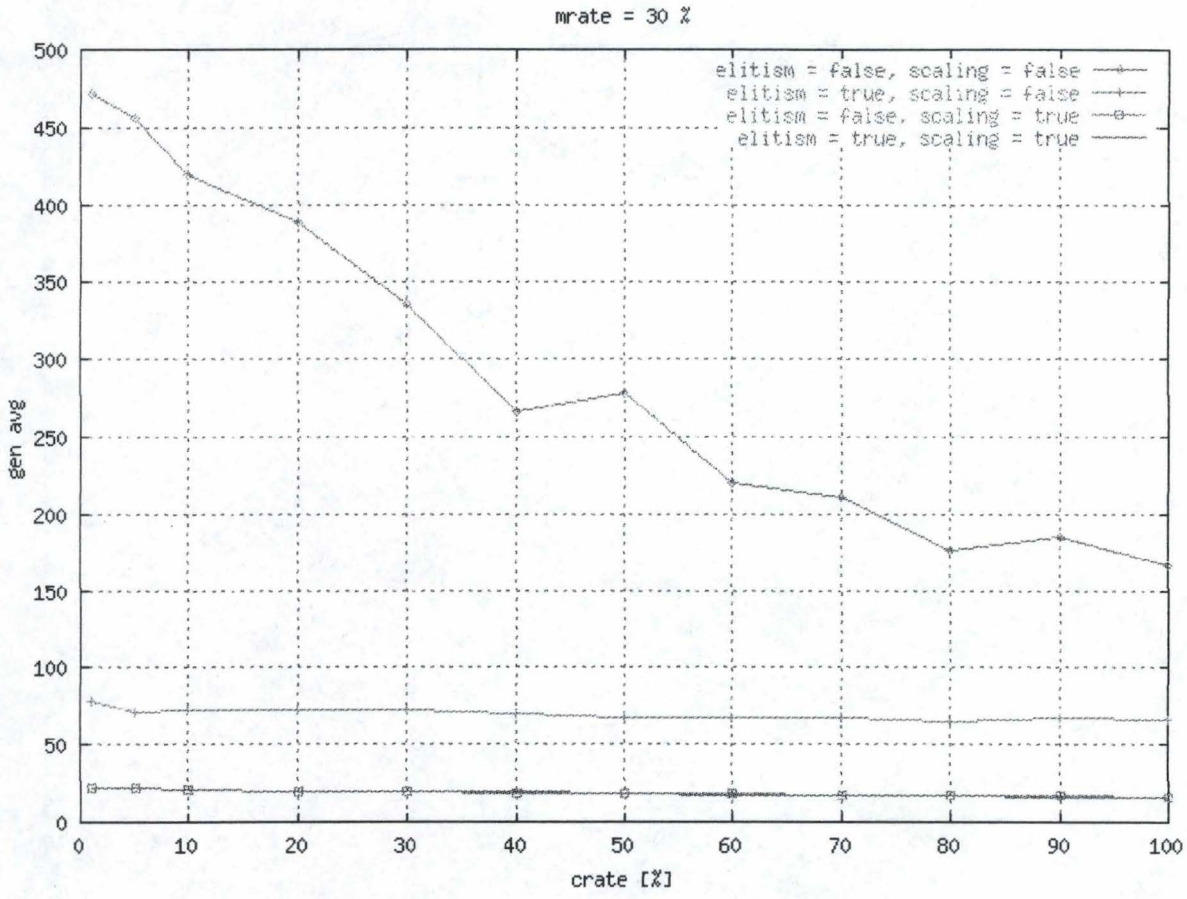
Dans les tableaux 7 et 8, on a alterné l'activation entre le scaling et l'élitisme. On remarque que le fait d'utiliser les deux ensemble (les deux sont activés) améliore sensiblement les performances

(comme on l'a déjà vu précédement), mais surtout le scaling qui joue le plus grand rôle dans cette amélioration. Comme si le scaling (le changement d'échelle) opère comme un brassage des individus de la population avant chaque reproduction. Vu la nature stochastique ("orientée" et tout de même par la mesure de gain) de la reproduction (sélection, croisement- crossover- et mutation), le fait de brasser les individus redistribue les cartes et augmente sensiblement les chances d'arriver très vite à la meilleure solution -meilleur chromosome ou individu- (cf.Annexe.3.L'évolution biologique et l'évolution des espèces). Cet effet est accentué par l'élitisme où on reporte les meilleurs individus de chaque génération à la suivante. On a effectué l'experimentation pour deux taux différents de mutation, pour être plus sûr de la conclusion.

- **mutation (mrate) = 30%**

crate	1%	5%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
gen. ave. Elit=F Sca=F	472	456	419	389	336	266	278	221	211	177	185	167
gen. ave. Elit=T Sca=F	79	71	72	72	72	70	67	68	67	65	67	66
gen. ave. Elit=F Sca=T	22	22	21	20	19	18	18	18	17	17	17	16
gen. ave. Elit=T Sca=T	22	22	21	20	19	19	18	17	17	17	16	16

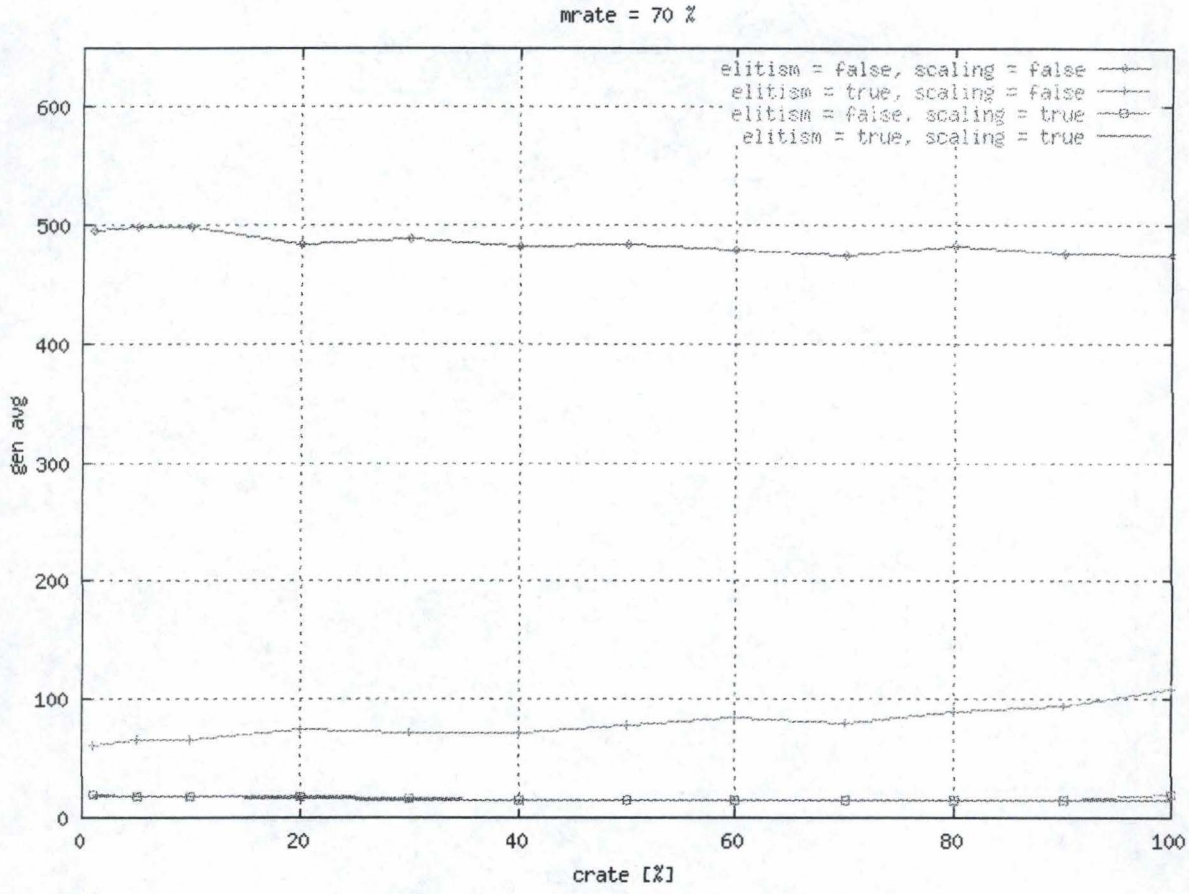
Tableau 7.



- **mutation (mrate) = 70%**

crate	1%	5%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
gen. ave. Elit=F Sca=F	496	498	498	484	489	483	484	480	474	483	476	474
gen. ave. Elit=T Sca=F	60	66	65	75	71	71	78	85	80	90	94	108
gen. ave. Elit=F Sca=T	19	18	17	17	16	15	15	15	14	14	14	19
gen. ave. Elit=T Sca=T	17	17	17	16	15	15	15	15	14	14	14	14

Tableau 8.



Le changement de la taille de la population accélère aussi la vitesse de l'algoithme. Avec une population de 25 individus, et un taux de mutation de 85% , l'algorithme prend en moyenne plusieurs cycles pour trouver une solution optimale - mais le traitement de chaque génération est si rapide à un point que la performance, en sa totalité, est améliorée (cf. tableau 9, figure 9):

Les paramètres :
Pop. Size: 25
Test Size: 500
Crossover: true (100%)
Mutation: true (85%)
Scaling: true
Elitism: true

Les résultats :
gen average = 17.71
gen minimum = 10
gen maximum = 34

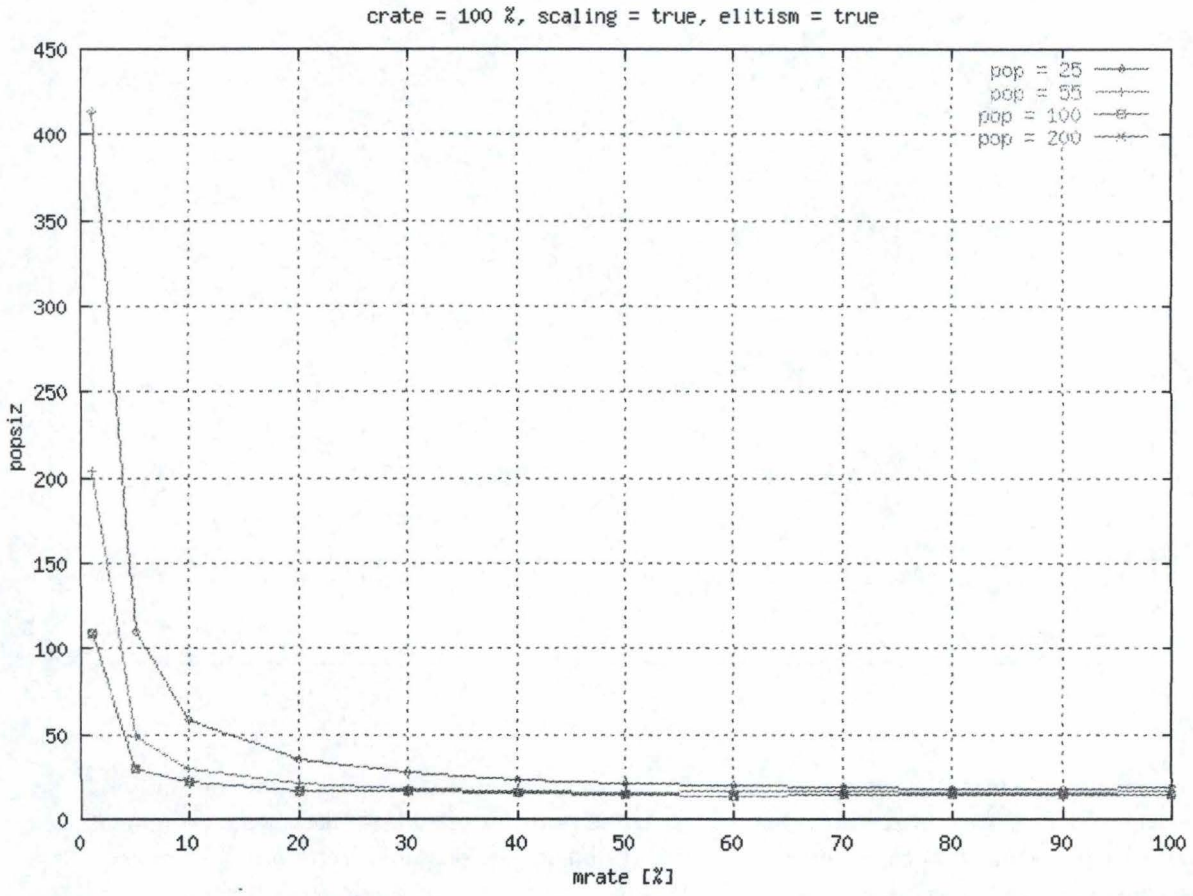
tick average = 85.13
tick minmum = 50
tick maximum = 165

- **crossover (crate) = 100%, scaling = true, elitism = true**

mrate popsiz	1%	5%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
25	414	110	58	35	28	23	21	20	19	18	18	19

35	310	72	45	27	22	20	18	17	16	15	17	18
55	204	48	30	21	18	16	15	16	16	15	15	16
75	147	36	25	19	17	15	14	14	14	14	14	15
95	119	33	22	17	16	15	14	14	13	13	13	14
100	109	30	22	17	16	15	14	13	14	14	14	14
200	60	23	18	16	15	14	14	13	13	13	13	13
500	31											12
1000	23											

Tableau 9.



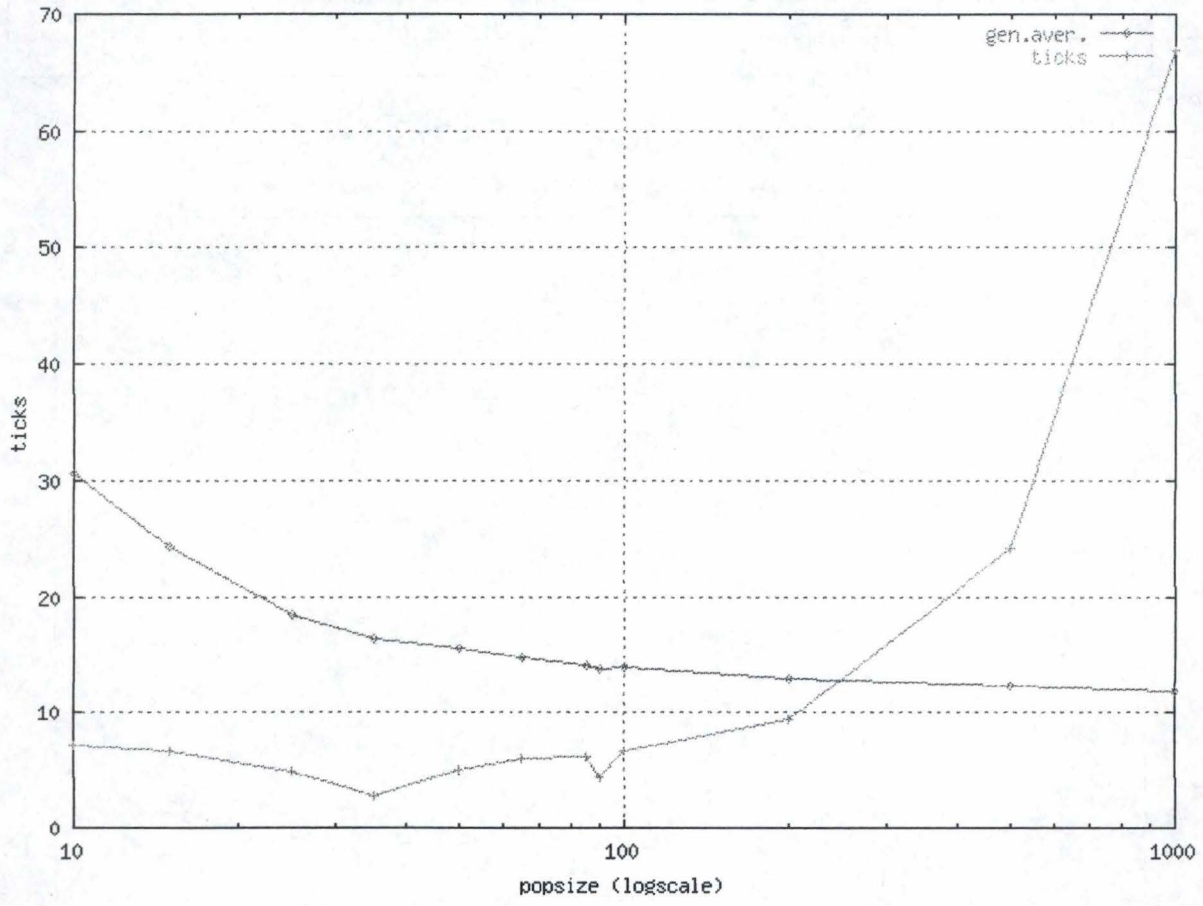
Il y a quatre remarques à faire: premièrement, le nombre moyen de générations avant la convergence diminue constamment avec l'augmentation de la taille de la population. Deuxièmement, cette diminution du nombre moyen est beaucoup plus significative pour des petits taux de mutation (1%-30%)que pour des grands taux. Troisièmement, on remarque un certain ralentissement de cette diminution à partir d'une certaine taille;on a du doubler la taille de 500 à 1000 pour gagner 26% (de 31 à 23), tandis que pour des petites tailles on a augmenté de 1.57 la taille (de 35 à 55) et on a gagné 35% une diminution (de 310 à 204). Enfin, on remarque un certain barrage qu'on n'arrive pas à le franchir autour des valeurs 12-13, et d'ailleurs pour cette raison on n'a pas marqué les valeurs correspondants aux tailles 500 et 1000.

Dans le tableau 10, on s'est intéressé en plus au temps, en ticks, que prend l'AG avant la convergence, et cela en fonction de la taille de la population. Donc la même chose que le tableau 9., mais en observant la relation entre la taille, le temps (en plus) et le nombre moyen de générations.

- **crate =100%, mrate = 85%, scaling = true, elitism = true**

popsiz	10	15	25	35	50	65	85	90	100	200	500	1000
gen. aver.	30.7	24.31	18.44	16.43	15.55	14.73	14.02	13.82	13.93	13.03	12.36	11.88
ticks	7.2	6.6	4.9	2.8	5	6	6.1	4.4	6.6	9.4	24.2	67

Tableau 10.



On remarque que en ce qui concerne le nombre moyen de générations, il y a une diminution constante, mais relativement faible -même très faible pour les grandes tailles- avec l'augmentation de taille de la population, ce qui est moins évident pour les ticks. Ainsi, cette diminution est constatée jusqu'à une certaine taille (± 40 individus), puis le temps augmente et même sensiblement pour les grandes tailles, ce qui peut être expliqué par le fait que ce qu'on gagne par l'effet de grande taille -et donc plus riche et plus variée-d'un côté, est largement absorbé par les cycles de plus en plus longs de l'évaluation et, surtout, de la reproduction.

Dans ce problème, l'AG cherchait une solution unique parmi plus que 4 milliards possibles- et il peut le trouver en seulement quelques secondes, et en ne testant ,en moyenne, qu'environ - seulement - 500 valeurs d'input.

Avec l'optimisation de l'output de *Blackbox*, on a montré les principes de base, les points essentiels et les caractéristiques des AG, et on a démontré leurs forces dans plusieurs domaines ; cependant il existe des manières et des méthodes (comme par exemple, les AG guidés par la connaissance, l'utilisation d'opérateurs génétiques différents du crossover et de la mutation, l'hybridation avec des autres méthodes d'optimisation, ou enfin l'implémentation des AG sur des ordinateurs parallèles) pour améliorer encore leurs performances et combler leurs lacunes. Examiner en détail chacune d'elle nous emmènerait au-delà de l'objectif de ce travail.

6. Recommandations

Les recommandations sont souvent les résultats des études empiriques des AG, et qui étaient souvent exécutés avec du codage binaire. Donc, voici quelques recommandations concernant les paramètres, les opérateurs et le codage.

- **Le taux de crossover**

Le taux de crossover doit être, généralement, élevé, autour de 80% - 95% . Cependant certains résultats de recherches montrent pour certains problèmes un taux autour de 60% est meilleur.

- **Le taux de mutation**

De l'autre côté, le taux de mutation doit être très bas. Les meilleurs taux signalés tournent autour de 0.5% - 1% .

- **La taille de la population**

Cela peut être surprenant, mais une très grande taille de population n'améliore pas habituellement la performance de l'AG (dans le sens de la vitesse de trouver une solution). Une bonne taille est autour de 20 -30 ; cependant, on a rapporté que parfois des tailles de (50 - 100) sont meilleures. Cretaines recherches montrent aussi, que la meilleure taille de population est une quantité qui dépend du codage, surtout de la longueur des chaînes - ou des chromosomes - qu'on veut encoder.

Cela veut dire que, si vous avez un chromosome de 32 bits, la population devrait être, disons, 32, mais sûrement 2 fois plus que la meilleure taille de population avec un chromosome de 16 bits.

- **La sélection**

La sélection de base (la roue de loterie) peut être utilisée, mais parfois la sélection par rangement est meilleure. Il y a des autres méthodes plus sophistiquées, qui changent les paramètres de sélection durant l'exécution de l'AG. Fondamentalement, ils se comportent comme le recuit simulé; mais sûrement l'élitisme doit être utilisé (si on n'utilise pas autre méthode pour sauver la meilleure solution trouvée) . On peut aussi utiliser la sélection par état-stable

- **Les types de crossover et de mutation**

Les opérateurs dépendent du type du codage et du problème .

- **Le codage**

Le codage dépend de la nature du problème et aussi de sa taille. Goldberg, dans son excellent livre ([GOL 91]) propose deux principes de base pour choisir un codage :

1) *Le principe de la pertinence des briques élémentaires (blocs de construction)* : L'utilisateur doit sélectionner un codage de façon à ce que les schèmes (schémas) courts et d'ordre faibles soient pertinents por le problème sous-jacent, et relativement indépendants des schèmes aux autres positions intanciés

2) *Le principe des alphabets minimaux* : L'utilisateur doit choisir le plus petit alphabet qui permette une expression naturelle du problème (pour plus d'explication, on peut se référer au livre de Goldberg.

Bibliographie

[APO 82] APOSTOL T.M., Mathematical Analysis, Addison-Wesley, 1982.

[BEL 90]* BELEW R. K., McINENREY, and SCHRAUDOLPH, Evolving Networks: Using the Genetic Algorithms with Connectionist Learning. CSE Technical Report CS90-174, University of California, 1990.

[BIN 78] BINDRE E., La Génétique Des Populations, que-sait-je?,PUF,1978.

[BLU 92] BLUM A., Neural Networks in C++, John Wiley, 1992.

[BUI 86] BUICAN D., La Génétique et l'Evolution, que-sait-je?,PUF,1986.

[DEV 99] DEVILLE Y., Calculabilité, Licence et maîtrise en informatique, FUNDP, 1998-1999.

[DAV 91] DAVIS L., Handbook of Genetic Algorithms, VAN NOSTRAND REINHOLD,1991.

[GOL 91] GOLDBERG D.E., Genetic Algorithms in Search, Optimization, and Machine Learning, Addison-Wesley,1991.

[HAY 99] HAYKIN S., Neural Networks, second edition, Prentice-Hall, 1999.

[HOL 93] HOLLAND J.H., in "A New Era in Computation", Edited by N. METROPOLIS and G.-C.ROTA,MIT Press,1993.

[HOL 92]* HOLLAND J.H., Adaptation in natural and artificial systems (second edition), MIT Press, 1992.

[KOS 92] Kosko B., Neural Networks and Fuzzy Systems, Prentice-Hall, 1992.

[KOZ 91] Koza J.R., Genetic Programming, MIT Press, Cambridge, MA, 1991.

[KRI 94] KRISHNAN B., CIESIELSKI V. B.,2DELTA_GANN/ A new approach to training Neural Networks using Genetic Algorithms,In A.C.(Ed.) Proceedings of the Fifth Australian Conference on Neural Networks, University of Queensland, Brisbane, pp38-41, 1994.

[KTK 96] K. T. KO, K. S. TANG, C. Y. CHAN,K. F. MAN and S. KWONG, "Using Genetic Algorithms to design mesh networks," IEEE Computer, pp.56-61, Aug 1997.

[MAW 84] MAWHIN J., Introduction à l'analyse, UCL, CABAY, 1984.

[MIC 92] MICHALEWICZ Z., Genetic Algorithms + Data Structures = Evolution programs, Springer-Verlag,1992.

[NEL 91] NELSON M. Mc., ILLINGWORTH W. T., A Practical Guide to Neural Nets, Addison-Wesley, 1991.

[REN 95] RENDERS J.-M., Algorithmes génétiques et réseaux de neurones, HERMES,1995.

[RUM 92]* RUMELHART D.E. and McLELLAND J.L., Parallel Distributed Processing: Explorations in the microstructure of Cognition. MIT Press, Cambridge, Massachusetts, 1989.

[SCH 89]* SCHAFFER J. D., CARUANA R. A., ESHELMAN L. J., and DAS R., A study of control parameters affecting online performance of genetic algorithms for function optimization. In J. D. SCHAFFER(ed), Proceedings of the Third International Conference on Genetic Algorithms. San Mateo, Calif.; Morgan Kaufmann Publishers, 1989.

[SCH 92]* SCHAFFER J.D., WHITLEY D., and ESHELMAN L.J., Combinations of Genetic Algorithms and Neural Networks: A survey of the state of the art. In COGANN -92, International Workshop on Combinations of Genetic Algorithms and Neural
Networks, pp1-37, Baltimore, June 1992. IEEE Computer Society Press.

[SCO 96] SCOTT R.L., Genetic Algorithms in C++, M&T Books,1996.

[SYS 89]* SYSWERDA G., Uniform crossover in Genetic Algorithms. In J. D. SCHAFFER(ed), Proceedings of the Third International Conference on Genetic Algorithms. San Mateo, Calif.; Morgan Kaufmann Publishers, 1989.

[TAN 97]* K. T. KO, K. S. TANG and K. F. MAN, "Microwave Communication System designs using
Hierarchy Genetic Algorithm," In Proceedings Asia Pacific Microwave Conference, pp.29-32, Dec, 1997, Hong Kong.

[VEI 91]* VEITCH A.C. and HOLMES G., Benchmarking and fast learning in neural netowrks-results for backpropagation. Proceedings of the 2nd Australian Conference on Neural Networks, pp133-139, Sydney,1991.

[WAS 93] WASSERMAN P. D., Advanced Methods in Neural Computing, Van Norstand Reinhold, New York, 1993.

[WHI 91]* WHITLEY D., MATHIAS K.and FITZHORN P.. Delta coding: An iterative search for genetic algorithms. In Proceedings of the Fourth International Conference on Genetic Algorithms, pp77-84,San Mateo, Calif.; Morgan Kaufmann 1991.

[WHI 89a]* WHITLEY D, the GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In Proceedings of the Third International Conference on Genetic Algorithms, pp117-121,San Mateo, Calif.; Morgan Kaufmann, 1989.

[WHI 89b]* WHITLEY D.,HANSON T. Optimizing neural networks using faster more accurate genetic search. In Proceedings of the Third International Conference on Genetic Algorithms, pp391-396,San Mateo, Calif.; Morgan Kaufmann 1989.

[WOL 91] WOLPER P., Introduction à la calculabilité, InterEditions, 1991.

*: Les ouvrages et les articles indiqués par un astérisque ne sont pas, directement, utilisés dans le mémoire.

Les magazines scientifiques :

- Sciences et Avenir-février2000-Dossier Spécial Génétique-Dix Questions sur la Génétique.
- Scientific American-July1992-Genetic Algorithms-J.H. HOLLAND.
- Scientific American-January2001-Complexit Theory-Software-WAKEFIELD J. .
- La Recherche-septembre2000-Idées,Entretien avec G.M. EDELMAN- pour une approche darwinienne du fonctionnement cérébral.
- La Rechreche-novembre2000-Actualité des Sciences-Robots darwiniens.

Articles d'Internet :

- Genetic Algorithms-Marek Obitko-1998-http://cs.felk.cvut.cz/~xobitko/ga
- Abrief Introduction To Genetic Algorithms-Moshe Sipper-2000-http://lslwww.epfl.ch/~moshes/ga
- Traveling Salesman Problem Using Genetic Algorithms-Mchael Lalena-1998 http://www.lalena.com/ai/tsp/
- Genetic Algoritms : Concepts and Designs-Kim-Fung Man,Kit-Sang Tang-Sam Kwong-City University of Hong Kong-4/98-http://Sant.bradley.edu/~ienews/98_4/genet.htm
- Genetic Java-Dan Loughlin-6/96-http://www4.ncsu.edu/eos/users/d/dhloughl/public/stable.htm