UNIVERSITÉ DE GENÈVE
Département d’ informatique

FACULTÉ DES SCIENCES
Prof. P. Zanella
Dr. B. Chopard

Genetic Programming
Methodology, Parallelization and Applications

THÈSE

présentée à la Faculté des sciences
de l’Université de Genève
pour obtenir le grade de
Docteur ès sciences mention Informatique

par

Mouloud OUSSAIDÈNE
de

Tizi-Ouzou (*Kabylie*), Algérie

Thèse N^{o} 2888

GENÈVE
Atelier de reproduction de la section de physique
1997

Remerciements

J’aimerais adresser mes remerciements au Professeur Paolo Zanella pour avoir accepté de diriger cette thèse en m’accueillant dans son équipe de recherche et m’offrir ainsi la chance d’entreprendre ce travail.

Mes remerciements vont au Professeur Christian Pellegrini pour m’avoir fait l’honneur de participer à mon comité de thèse et pour l’intérêt qu’il a porté à mes travaux.

J’aimerais remercier le Docteur Bastien Chopard pour m’avoir offert son entière disponibilité tout au long de ce travail. J’apprécie ses qualités aussi bien scientifiques qu’humaines. Son esprit si prompt à saisir l’essentiel d’un problème, son ouverture d’esprit fort agréable ainsi que sa volonté d’aller toujours plus loin ont permis à cette thèse d’atteindre une qualité reconnue au niveau international. J’aimerais lui exprimer ici ma profonde gratitude pour sa tolérance et tout le soutien qu’il m’a apporté durant cette recherche.

Docteur Marco Tomassini est l’instigateur potentiel de ce projet. Je le remercie tout particulièrement pour la confiance qu’il a mise en moi quant à la réalisation et l’aboutissement de ce travail. Il a toujours apporté de nouvelles idées dans chacune de nos réunions et a su créer une atmosphère de recherche stimulante et fructueuse au sein de notre groupe. Nous avons entretenu des rapports aussi bien professionnels qu’amicaux, et je tiens à le remercier pour tous les conseils précieux qu’il m’a prodigués et la compétence dont il m’a fait part, entre autres, en algorithmes génétiques qui constituent les fondements de cette thèse.

Docteur Olivier Pictet m’a initié au domaine financier et m’a apporté une expertise inestimable. Je le remercie pour les données financières qu’il a mises à ma disposition et offrant ainsi à cette thèse la possibilité de sortir du monde académique et de traiter un problème du monde réel aussi complexe que le marché financier.

Table Of Contents

Résumé

Les algorithmes évolutionnistes, dont les algorithmes génétiques, sont des techniques de recherche adaptatives s'inspirant des mécanismes de sélection naturelle.
Il s'agit de procédures itératives qui se déroulent comme suit:

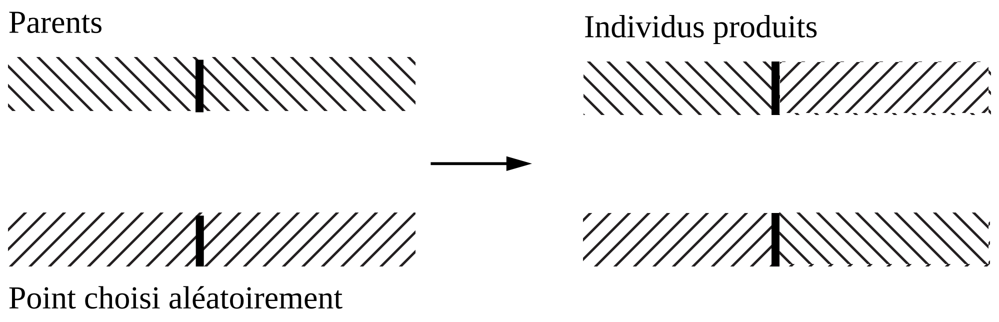
On génère une population initiale de manière aléatoire ou heuristique, dans laquelle chaque individu représente une solution possible d'un problème donné a travers un codage opportun conduisant à une représentationen chaîne de longueur finie, par exemple une chaîne binaire.
A chaque pas de l'itération, appellé également génération, les individus sont évalués en fonction d'un critère d'adaptation fixé.

Dans une première phase, les individus sont choisis avec une probabilité proportionnelle à leur valeur d'adaptation (ou coût) , afin de favoriser les meilleurs d'entre eux. Pour reformer une nouvelle population de taille constante et générer de nouveaux points dans l'espace de recherche, on utilise les opérateurs dits génétiques.

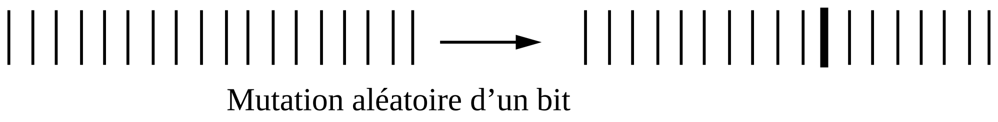
Un algorithme génétique standard se formule donc comme suit:

```
- Générer aléatoirement une population d'individus de taille donnée.
- Répéter
           Évaluation: Affecter un coût à chaque individu.
            Sélection: Établir une liste de paires d'individus. La sélection est basée sur les
                       coûts associés aux individus.
         Reproduction: Appliquer les opérateurs génétiques a toutes les paires d'individus. Les
                       individus produits représentent la nouvelle population.
-Jusqu'à condition de terminaison.
```

Il y a deux types principaux d'opérateurs génétiques: l'hybridation (crossover) et la mutation.
L' opérateur crossover est une recombinaison de deux individus parents produisants deux nouveaux individus. Cette opération se fait en prenant dans chacun des deux parents des sous-chaînes complémentaires de leur code "génétiques" , comme indiqué ci-dessous:



La mutation consiste en un changement aléatoire d'un symbole dans la chaîne codant l'individu.



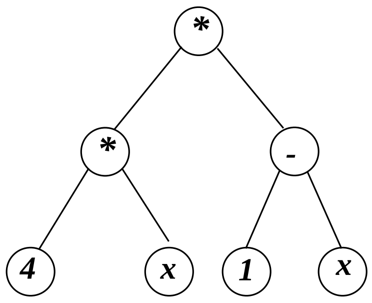
La mutation sert à éviter que la population ne se fixe trop rapidement sur des optima locaux, phénomène que l'on nomme convergence prématurée.

Le cycle évaluation/sélection/recombinaison/mutation est réitéré jusqu'à ce qu'un critère d'arrêt soit atteint. Ceci peut survenir après un nombre maximal prédéterminé de générations ou à l'obtention d'une solution satisfaisante.

Il est à remarquer que les algorithmes évolutionnistes ont été appliqués avec succès à beaucoup de problèmes difficiles en optimisation de fonctions et en optimisation combinatoire, ainsi que dans l'inférence des règles de décision.

La programmation génétique est l'extension du modèle génétique d'apprentissage décrit précédemment à l'espace des programmes. Le concept peut être perçu comme l'application d'un algorithme génétique à un espace de recherche constitué d'expressions Lisp. Dans le contexte de la programmation génétique, un individu est appellé programme génétique (PG). Ainsi, les différentes phases d'un algorithme génétique classique opèrent sur des expressions Lisp (S-expressions) plutôt que sur des chaînes binaires de taille fixe.
Un PG est souvent représenté par un arbre, ce qui facilite aussi bien l'extension des opérateurs génétiques que l'évaluation du programme lui-même.

Par exemple, l'expression *(* (* 4 x) (- 1 x))* peut être représentée graphiquement comme suit:



Les noeuds fonctions de cet arbre sont (* , *, -) et les noeuds terminaux sont (*4, x, 1, x*). La racine de cet arbre correspond au premier noeud fonction apparaissant dans la S-expression (i.e *).
L'espace de recherche en programmation génétique correspond à toutes les combinaisons de fonctions créées récursivement à partir d'un ensemble de N_f fonctions

$$F = \{f_1, f_2, \ldots, f_{N_f}\}$$

et d'un ensemble de N_t terminaux

$$T = \{a_1, a_2, \ldots, a_{N_t}\} \quad .$$

L'ensemble, F, des fonctions est appellé *Function-Set* et l'ensemble, T, des terminaux est appellé *Terminal-Set*.

Le choix des ensembles F et T doit se faire de manière à ce que les S-expressions formées à partir de l'ensemble $F \cup T$ soient des solutions au problème à résoudre.
Par exemple, dans le cas de la logique binaire, l'ensemble F = {AND, OR, NOT} est suffisant pour réaliser n'importe qu'elle fonction Booléenne.

La création de chaque S-expression dans la population initiale commence à partir de la racine. Le noeud racine est souvent choisi comme fonction pour éviter l'obtention d'une expression constituée d'un seul terminal. Lorsqu'une fonction f de F est choisie comme noeud de l'arbre, alors r noeuds, où r est l'arité de f, sont crées à patir de ce noeud. Ainsi, pour chaque noeud fonction, un élément de l'ensemble $F \cup T$ est choisi aléatoirement comme argument de cette fonction.

Si une fonction est choisie comme argument, alors le processus continue récursivement. Si un terminal est choisi comme argument d'une fonction, alors il devient noeud terminal (feuille) de l'arbre et le processus s'achève.
Il est aisé de remarquer que cette construction nous garantie la validité syntaxique (mais pas forcément sémantique) des programmes génétiques.

L'extension de l'opérateur crossover à l'espace des S-expressions se fait comme suit. On commence par sélectionner deux PG et choisir un noeud sur chacun d'eux. Chaque noeud est, par définition, la racine d'un sous-arbre. Les deux sous-arbres sont extraits et interchangés.

Ainsi, si on choisit les noeuds 4 et 2 respectivement sur les PG suivants (les noeuds sont numérotés de gauche à droite)

*(+ 2 (* x (/ x 3))) et (+ (* x 4) 1)*

on obtient, comme nouveaux individus

*(+ 2 (* (* x 4) (/ x 3))) et (+ x 1)*

Dans le cas de l'opérateur mutation, un noeud terminal est échangé avec un autre noeud terminal, tandis qu'un noeud fonction n'est interchangeable qu'avec une autre fonction ayant le même nombre d'arguments.

Dans la programmation génétique, les fonctions et terminaux ne sont pas associés à des positions fixes dans l'arbre. En outre, il est relativement rare qu'un terminal ou une fonction disparaisse entièrement de la population. Par conséquent, en programmation génétique, la mutation n'a pas le rôle potentiel de restaurer la diversité dans la population, comparativement au cas de l'algorithme génétique classique.

Comme mentionné précédemment, les programmes génétiques générés sont syntaxiquement valides. Cependant, chaque fonction dans le Function-Set doit être bien définie de manière à accepter toute combinaison d'arguments. Autrement dit, il faut introduire la gestion des exceptions quand une fonction est indéfinie pour certaines valeurs de ses paramètres.
Par exemple, l'expression suivante PG = *(* (+ x 1) (/ x 0))* est syntaxiquement correcte, mais la division par zéro la rend sémantiquement incorrecte. Pour pallier à ce genre d'exceptions, la solution standard consiste à redéfinir la fonction de division de manière à retourner la valeur 1 si le dénominateur vaut 0.

Ces dernières années, la programmation génétique a suscité une attention particulière dans la communauté scientifique. Plusieurs travaux sont effectués dans l'optimisation des requêtes en bases de données, optimisation des modèles de marché financier, robotique, etc ...

Dans ce travail, nous présentons un système de programmation génétique destiné à s'exécuter sur des machines à passage de méssages. Le système est écrit en C++ et utilise les primitives PVM. L'utilisateur doit fournir une fonction qui associe une mesure de qualité à chaque programme génétique. Cette fonction est directement liée au problème à résoudre. Notre système offre une grande convivialité et l'aspect parallèle est entièrement transparent à l'utilisateur.

L'analyse de la compléxité d'une exécution séquentielle nous a montré que le temps total d'exécution est directement proportionnel à la taille du problème et que la phase d'évaluation consomme le plus de temps quand le problème nécessite plus de 1000 cas d'évaluations.
La parallélisation de la phase d'évaluation, en programmation génétique, se heurte au problème de la variabilité de la taille des programmes à évaluer. La taille d'un programme est définie par le nombre d'opérations arithmétiques nécessaires pour son évaluation.
En effet, ce problème équivaut à trouver un ordonnancement optimal de P tâches indépendentes sur $m < P$ machines. Ceci est un problème NP-difficile.

Nous proposons dans cette étude deux algorithmes pour équilibrer la charge sur les processeurs. Le premier algorithme consiste à répartir les tâches de manière circulaire: la tâche *j* est affectée au processor *j* MOD *m*. C'est un schéma statique d'équilibre de la charge. Le second algorithme est basé sur la taille des tâches à l'exécution. Les tâches sont d'abord triées de manière décroissante selon leur taille, et ensuite la distribution des tâches se fait de la plus grande vers la plus petite. A chaque itération, l'algorithme affecte la tâche courante au processeur le moins chargé. C'est un schéma dynamique d'équilibre de la charge.

Nous montrons qu'en l'absence de variabilité dans la taille des tâches, les deux algorithmes aboutissent aux mêmes performances. Par contre, dans le cas où il y a une grande diversité dans la taille des tâches, les performances du schéma dynamique sont meilleures que celles obtenues avec le schéma statique. Cette différence s'explique par le fait que la charge d'un processeur n'est pas déterminée par le nombre de tâches qui lui sont assignées, mais plutôt par la totalité d'instructions à exécuter. Nous montrons que notre implémentation peut délivrer un speedup presque linéaire pour des problèmes de grande taille.

La parallélisation de la phase d'évaluation ne pallie pas au problème de la convergence prématurée, phénomène caractérisant l'algorithme génétique conventionnel. Pour ce faire, notre système intègre le modèle à ilôts en faisant évoluer plusieurs populations en parallèle. Afin d'éviter qu'une sous-population ne converge rapidement vers un optimum local, des individus migrent entre les sous-populations selon la topologie d'anneau. L'effet de la migration est de réintroduire la diversité dans les sous-populations.

La programmation génétique a été appliquée avec beaucoup de succès à beaucoup d'applications dont la solution optimale est connue au préalable. Cependant, il est important de savoir si cette méthode s'applique aussi à des problèmes réels où les réponses ne sont pas connues et les données contiennent du bruit.

Nous présentons l'application de la programmation génétique à l'apprentissage des modèles d'investissement pour le marché financier. Les recommandations suggérées par ces modèles sont basées sur les fluctuations des prix dans le passé. L'historique des prix est résumé sous forme de variables appellées *indicateurs*. Les indicateurs que nous utilisons sont basés sur des moyennes mobiles. La forme la plus simple d'un modèle d'investissement peut être décrite par la règle suivante:

```
IF |I| > K THEN G := SIGN(I)                    (*)
ELSE G := 0
```

Où la variable I est un indicateur dont le signe et la valeur reflètent la tendance actuelle des prix, et K est une constante seuil (*break-level*).
Les recommandations du modèle sont les valeurs prises par la variable G. La valeur $G = +1$ correspond à un signal d'achat, $G = -1$ correspond à un signal de vente et $G = 0$ correspond à la position neutre. Comme l'utilisation d'un seul indicateur ne peut signaler tous les changements de tendance, il est important de combiner plusieurs d'entre eux afin d'obtenir une image globale du marché.

Dans cette étude, un modèle d'investissement est une combinaison logique de plusieurs règles ayant la forme (*). Les signaux retournés par ces règles sont combinés par des opérateurs logiques (AND, OR, NOT, IF). Un programme génétique est alors représenté par un arbre de décision dont les noeuds terminaux sont des indicateurs. Pour chaque prix, les indicateurs sont d'abord mis à jour, ensuite l'arbre est évalué pour obtenir la recommandation finale du modèle. Une transaction a lieu quand la position recommandée est différente de la position courante.

La qualité d'un modèle d'investissement n'est pas basée uniquement sur la maximisation du rendement (*return*) , mais aussi sur la minimisation du risque. Une mesure du risque induit par un modèle financier est le calcul de la variance du rendement sur toutes les transactions opérées. Afin de mesurer la qualité d'un modèle d'investissement, nous utilisons la différence entre le rendement moyen et la variance du rendement lui-même. Ainsi, un 'bon' modèle est un modèle ayant un rendement positif constant sur chaque transaction.

Dans cette étude, les prix utilisés sont des données réelles. La période d'optimisation, contenant des données horaires, couvrant 7 monnaies (*GBP/USD*, *USD/DEM*, *USD/ITL, USD/JPY, USD/CHF, USD/FRF, USD/NLG*), commence le *1er Janvier 1987* et se termine le *31 décembre 1994*.

Pour pallier au problème de sur-apprentissage, les précautions suivantes ont été prises:
- chaque modèle est testé sur les 7 monnaies;
- chaque série de prix est subdivisée en périodes alternées optimisation/test;
- et la mesure de qualité d'un modèle d'investissement tient compte du risque sous-jacent.

Nous montrons que les modèles obtenus sont robustes et offrent un rendement moyen dépassant les 5% du montant de la transaction. Nos résultats attestent que la programmation génétique constitue un axe de recherche prometteur pour l'optimisation des modèles d'investissement financier.

1. Fundamentals of Genetic Algorithms

1.1 Introduction

Genetic Algorithms (GAs) are adaptative methods which may be used to solve search and optimisation problems. They are inspired by the genetic processes of biological organisms. Over many generations, natural populations evolve according to the principles of natural selection and “survival of the fittest”, first clearly stated by Charles Darwin in *The origin of Species*. By mimicking this process, genetic algorithms are able to “evolve” solutions to real world problems, if they have been suitably encoded.

The basic principles of GAs were first laid down rigourously by Holland [5], and are well described in many texts ([2] , [4] and [10]).

In nature, individuals in a population compete with each other for resources. Also, members of the same species often compete to attract a mate. Those individuals which are most succesful in surviving and attracting mates will have relatively large number of offspring. Poorly performing individuals will produce few or even no offspring at all. This means that the genes from the highly adapted, or “fit” individuals will spread to an increasing number of individuals in each successive generation. The combination of good characteristics from different ancestors can sometimes produce “superfit” offspring, whose fitness is greater than that of either parent. In this way, species evolve to become more and more well suited to their environment.

GAs use a direct analogy of natural behaviour. They work with a *population* of “individuals”, each representing a possible solution to a given problem. Each individual is assigned a “fitness score” according to how good a solution to the problem it is. In nature, this is equivalent to assessing how effective an organism is at competing for resources. The highly fit individuals are given opportunities to “reproduce”, by “cross breeding” with other individuals in the population. This produces new individuals as “offspring”, which share some features taken from each “parent”. The least fit members of the population are less likely to get selected for reproduction, and so do not survive.

A whole new population of possible solutions is thus produced by selecting the best individuals from the current “generation”, and mating them to produce a new set of individuals. This new generation contains a higher proportion of the characteristics possessed by the good members of the previous generation. In this way, over many generations, good characteristics are spread throughout the population, being mixed and exchanged with other good characteristics as they go. By favouring the mating of the more fit individuals, the most promising areas of the search space are explored. If the GA has been designed well, the population will *converge* to a good or even optimal solution to the problem.

1.2 Basic Principles

The standard GA can be represented as shown in figure 1.
Before a GA can be run, a suitable coding (or representation) for the problem must be devised. We also require a *fitness function*, which assigns a figure of merit to each coded solution. During the run, parents must be *selected* for reproduction, and *recombined* to generate offspring. These aspects are described below.

```
BEGIN /* genetic algorithm */

  Create an initial random population
  Evaluate the fitness of each individual in the population

  WHILE NOT termination condition DO
  BEGIN /* produce new generation */

       Select fitter individuals from current generation
       Recombine individuals
       Mutate individuals with low probability
       Evaluate the fitness of the new individuals
       Insert the new individuals in next generation

  ENDWHILE
END
```

fig 1: A traditional Genetic Algorithm

1.3 Coding

It is assumed that a potential solution to a problem may be represented as a set of parameters. These parameters (known as *genes*) are joined together to form a string of values (often referred to as a *chromosome*). Holland [5] first showed that a convenient problem representation is to use a binary alphabet for the string.
For example, if our problem is to maximise a function of three variables, *F(x, y, z)*, we might represent each variable by a 10-bit binary number. Our chromosome would therefore contain three genes, and consist of 30 binary digits.

1.4 Fitness function

A fitness function must be devised for each problem to be solved. Given a particular chromosome, the fitness function returns a single numerical “fitness”, or “figure of merit”, which is supposed to be proportional to the “utility” or “ability” of the individual which that chromosome represents. For many problems, particularly function optimisation, it is obvious what the fitness function should measure. Sometimes, it should just be the value of the function. The general rule in constructing a fitness function is that it should reflect the value of the chromosome in some “real” way. However, the real value of a chromosome is not always a useful quantity for guiding genetic search. In combinatorial optimisation problems, such as the construction of school timetables, most points in the search space often represent invalid chromosomes - and hence have zero “real” value. Nevertheless, special-purpose codings and genetic operators can be devised such that all generated solutions are viable [10].

1.5 Reproduction

During the reproduction phase of the GA, individuals are selected from the population and recombined, producing offspring which will comprise the next generation. Parents are selected randomly from the population using a scheme which favours the most fit individuals. Good individuals will probably be selected several times in a generation, poor ones may not be at all.
Having selected two parents, their chromosomes are *recombined*, typically using the mechanisms of *crossover* and *mutation*.

- Crossover takes two individuals, and cuts their chromosome strings at some randomly chosen position, to produce two “head” segments, and two “tail” segments. The tail segments are then swapped over to produce two new full length chromosomes (figure 2). The two offspring each inherit some genes from each parent. This is known as *single point* crossover.
Crossover is not usually applied to *all* pairs of individuals selected for mating. A random choice is made, where the likelihood of crossover being applied is typically between 0.6 and 1.0. If crossover is not applied, offspring are produced simply by duplicating the parents. This gives each individual a chance of preserving its genes without the disruption of crossover.

- Mutation is applied to each child individually after crossover. It randomly alters each gene with a small probability. Figure 3 shows the fifth gene of the chromosome being mutated.

The traditional view is that crossover is the most important of the two techniques for rapidly exploring a search space. Mutation provides a small amount of random search, and helps ensure that no point in the search space has a zero probability of being examined.

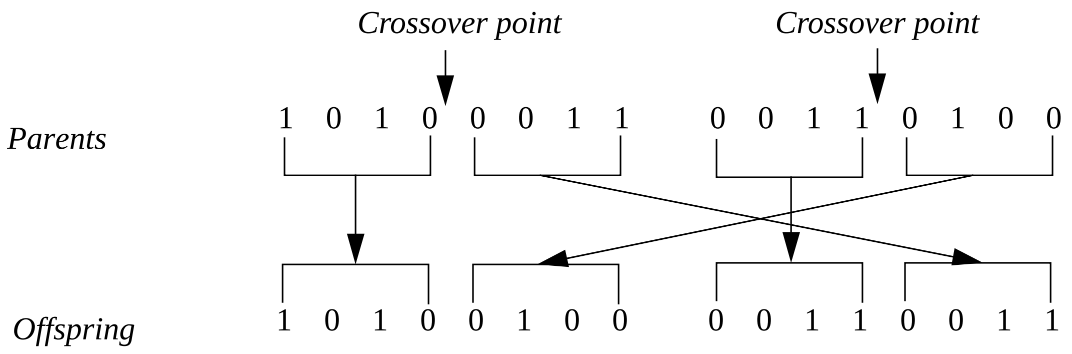


fig 2: Single-point Crossover

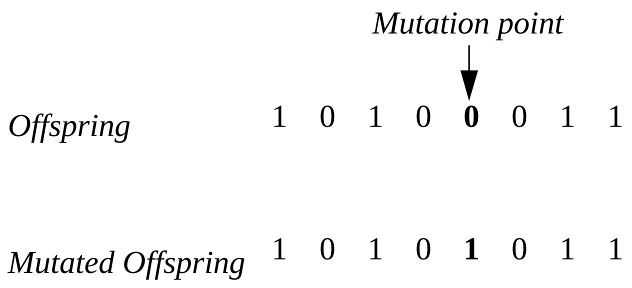


fig 3: A single mutation

Many different crossover techniques have been devised, often involving more than one cut point. DeJong [3] investigated the effectiveness of multiple-point crossover, and concluded that 2-point crossover gives an improvement, but that adding further crossover points reduces the performance of the GA. In fact, the problem with adding additional crossover points is that building blocks are more likely to be disrupted. However, an advantage of having more crossover points is that the problem space may be searched more thoroughly.

1.6 2-point crossover

In 2-point crossover, and multi-point crossover in general, chromosomes are regarded as *loops* formed by joining the ends together, rather than linear strings (figure 4). A 2-point crossover operator uses two randomly chosen cut points. Strings exchange the segments that falls between these two points. When viewed in this way, 1-point crossover is a special case of 2-point crossover where one of the cut points fixed at the start of the string.

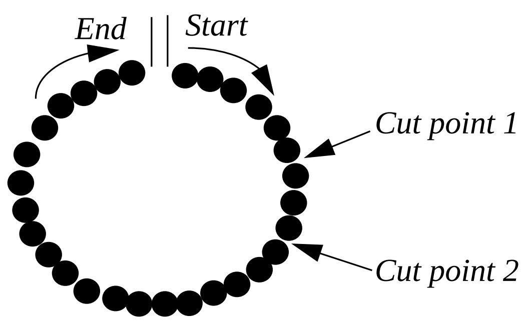
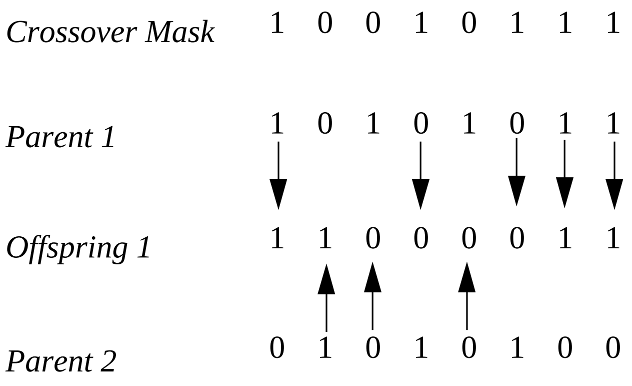


fig 4: Chromosome viewed as a loop

1.7 Uniform crossover

With *uniform crossover*, each gene in the offspring is created by copying the corresponding gene from one of the two parents, chosen according to a randomly generated *crossover mask*. Where there is a 1 in the crossover mask, the gene is copied from the first parent, and where there is a 0 in the mask, the gene is copied from the second parent, as shown in figure 5. The process is repeated with the parents exchanged to produce the second offspring. A new crossover mask is randomly generated for each pair of parents.
Offspring therefore contain a mixture of genes from each parent.



Crossover Mask	1	0	0	1	0	1	1	1
Parent 1	1	0	1	0	1	0	1	1
Offspring 1	1	1	0	0	0	0	1	1
Parent 2	0	1	0	1	0	1	0	0

fig 5: Uniform crossover

- Goldberg [4] describes a rather different crossover operator, *partially matched crossover* (PMX), for use in order-based problems. In an order based problem, such as the travelling salesman problem, gene values are fixed, and the fitness depends on the order in which they appear. PMX operator does not cross the values of the genes, but the order in which they appear. Offspring have genes which inherit ordering information from each parent. This avoids the generation of offspring which violate the problem constraints.

1.8 Convergence

If the GA has been correctly implemented, the population will evolve over successive generations so that the fitness of the best and the average individual in each generation increases towards a near-optimal solution. *Convergence* is the progression towards increasing uniformity. A gene is said to have converged when 95% of the population share the same value [3]. The *population* is said to have converged when all of the genes have converged. As the population converges, the average fitness will approach that of the best individual.

1.9 Parent selection techniques

Parent selection is the task of allocating reproductive opportunities to each individual. In principle, individuals from the population are copied to a “mating pool” , with highly fit individuals being more likely to receive more than one copy, and unfit individuals being more likely to receive no copies. The size of the mating pool is equal to the size of the population. Then, pairs of individuals are taken out of the mating pool at random, and mated. This process is repeated until the mating pool is exhausted.
The behaviour of the GA depends very much on how individuals are chosen to go into the mating pool.
There are a number of methods to do selection.

(a) - *Fitness-proportionate* selection assigns each individual structure *i* in the population a probability of selection $p_s(i)$, according to the ratio of the individual fitness to overall population fitness:

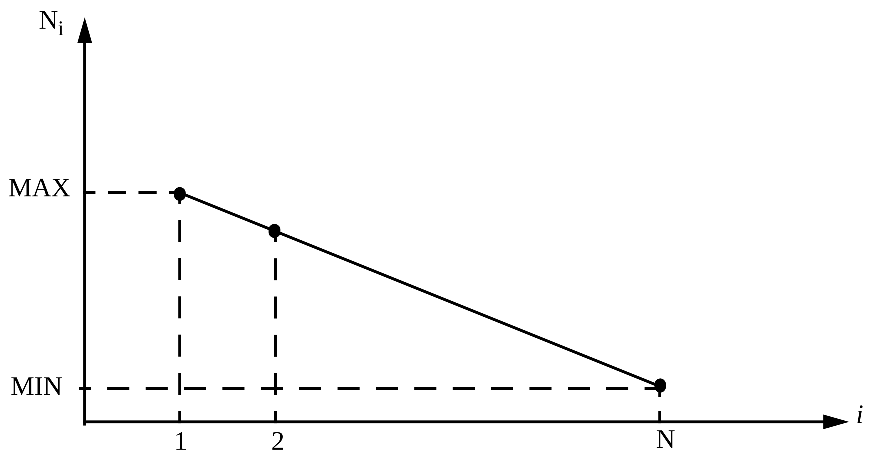
$$p_s(i) = f(i) \;/\; \left(\sum_{j=1}^{PopulationSize} f(j) \right)$$

Then it selects (with replacement) a total of *PopulationSize* individuals for further genetic processing, according the distribution defined by p_S . The simplest variant of fitness-proportionate selection, *roulette-wheel* selection [4] [11], chooses individuals through *PopulationSize* simulated spins of a roulette wheel. The roulette wheel contains one slot for each population element. The size of each slot is directly proportional to its respective $p_s(i)$. Note that the population members with higher fitnesses are likely to be selected more often than those with lower fitnesses. However, with this method the genes from a few comparatively highly fit individuals may rapidly come to dominate the population, causing it to converge on local optimum. Once the population has converged, the ability of the GA to continue to search for better solutions is effectively reduced: crossover of almost identical chromosomes does not introduce diversity. Only mutation remains to explore entirely new regions, and this simply performs random search.

(b) - Some variations on selection methods which do not allocate trials proportionally to fitness are *fitness ranking* selection and *tournament* selection.
Rank-based selection consists of allocating reproductive trials according to the rank of the

individual strings in the population rather than by individual fitness relative to the population average.
Ranking is an effort to slow down premature convergence. One cause of premature convergence may be “super individuals” that have an unusually high fitness ratio and thus dominate the search process. Ranking acts as a smoothing function which reduces the effect of exagerated differences in fitness.
However, the most serious objection to ranking is that it violates the schema theorem: the average of the rank of the individuals that sample a particular schemata does not correspond to the rank of the schema’s average fitness [14].

- Baker [1] used a linear curve for the allocation of trials.
The curve is defined by two points (1 , MAX) and (N , MIN) where MAX and MIN are respectively the number of trials allocated to the top and last ranked individuals in the population of size N.



That is an individual of rank *i* will receive a number of offspring which can be expressed as

$$N_i = \frac{MIN - N\ \ MAX}{1 - N} + \frac{MAX - MIN}{1 - N} i$$

and consequently will be selected with the probability

$$p_s(i) = \frac{N_i}{N}$$

Or

$$p_s(i) = \frac{1}{N}\left[MAX - \frac{(MAX - MIN)\ (i - 1)}{N - 1}\right]$$

The parameter MAX is a user defined value. Baker found that a value of MAX = 1.1 is suitable to prevent undesirable convergence.

The total of the individual allocation of trials should be equal to the population size N. This can be expressed as follows.

$$\sum_{i=1}^{N} N_i = N$$

yielding

$$MIN + MAX = 2$$

- Whitley [14] used a ranked-based allocation of reproductive trials which is similar to Baker’s linear ranking.

(c) - In *tournament* selection, a group of individuals is selected from the population with a uniform random probability distribution. The fitness value of each member of this group are compared and the actual best is selected. The size of the group is given by the *TournamentSize*. The most common variation, *binary tournament* selection, uses *TournamentSize* = 2.

The following pseudo-code describes the principle of this selection method. The procedure *Tournament* first forms the group of individuals, Σ, and then returns the individual whose fitness is maximum.

```
individual Tournament()
BEGIN

    $\Sigma = \emptyset$

    FOR i = 1 to TournamentSize DO
    BEGIN
        generate a random number $j \in [1, PopulationSize]$
        add individual j to the tournament set : $\Sigma = \Sigma \cup j$
    ENDFOR
    $j^* = \{j | fitness(k) \leq fitness(j), k \in \Sigma\}$
    return individual $j^*$
END
```

1.10 Generation gaps and steady-state replacement

The generation gap is defined as the proportion of individuals in the population which are replaced in each generation. Traditionally, a generation gap of 1 (the whole population) is replaced in each generation. Syswerda [12] introduced *steady-state* replacement where only a few (typically two) individuals, in each generation, are replaced.

In steady-state replacement, two parents are selected for reproduction and produce offspring that are immediately placed back into the population. The offspring do not replace parents, but

rather the least fit (or some relatively less fit) member of the population.
The essential difference between a generational GA and a steady-state GA, is that population statistics (such as average fitness) are recomputed after each mating in a steady-state GA and the new offspring are immediately available for reproduction. The advantage is that the best points found in the search are maintained in the population.

1.11 Schemata

While a GA on the surface processes strings, it implicitly processes schemata, which represent similarities between strings. A GA can not, as a practical matter, visit every point in the search space. It can, however, sample a sufficient number of hyperplanes in highly fit regions of the search space. Each such hyperplane corresponds to a set of highly fit, similar substrings.

A *schema* is a string of total length *l* (the same overall length as the population's strings), taken from the alphabet { 0 , 1 , * } , where '*' is a wild-card or "don't care" character. Each schema represents the set of all binary strings of length *l* whose corresponding bit-positions contain bits identical to those '0' and '1' bits of the schema. For example, the schema, 10**1, represents the set of five-bit strings, { 10001, 10011, 10101, 10111 }. Schemata are also called *similarity subsets* because they represent subsets of strings with similarities at certain, fixed bit-positions.
Two properties of schemata are their *order* and *defining length*. Order is the number of fixed bit-positions (non-wild-cards) in a schema. The defining length of a schemata is based on the distance between the first and last bits in the schema with value either 0 or 1. If I_x is the index of the rightmost occurrence of either a 0 or 1 and I_y is the index of the leftmost occurrence of either a 0 or 1, then the defining length is $I_x - I_y$.
For example, the following schema is of order 4, written $o(****1**0**10**) = 4$, and has a defining length of 7, written $\delta(****1**0**10**) = 12 - 5 = 7$. Each string in the population is an element of 2^l schemata.

1.12 Building blocks and schema theorem

Building blocks are low-order, short defining-length, highly fit schemata, where the fitness of a schema is defined as the average fitness of the elements it contains. Building blocks represent similarities (between strings) that are significant to the GA's solution of a particular problem.
Selection chooses strings with higher fitnesses for further processing. Hence strings that are members of highly fit schemata are selected more often. Crossover infrequently disrupts schemata with shorter defining lengths, and mutation infrequently disrupts lower order schemata. Therefore, highly fit, short defining length, low-order schemata, otherwise known as building blocks, are likely to proliferate from generation to generation. From this fact comes the claim that GAs process building blocks, also known as *useful schemata*. Holland [6] estimates that while a GA processes *n* strings each generation, it processes on the order of n^3 useful schemata. He called this phenomena *implicit parallelism*. Whitley [15] showed that to make this estimation reasonable, n must be chosen with respect to the string length.

Let $m(H, t)$ be the number of instances of schema H present in the population at generation t. We calculate the expected number of instances of H at the next generation, or $m(H, t+1)$, in terms of $m(H, t)$. The canonical GA assigns a string a selection probability directly proportional to fitness. Using fitness-proportionate selection, H can expect to be selected $m(H, t) \cdot f(H, t) / \langle f(t) \rangle$ times, where $\langle f(t) \rangle$ is the average population fitness and $f(H, t)$ is the average fitness of those strings in the population that are elements of H.
That is the number of strings in the population grows as the ratio of the fitness of the schema to the average fitness of the population. If we assume that a schema H remains above average by ε i.e $f(H, t) = \langle f(t) \rangle + \varepsilon \langle f(t) \rangle$, then $m(H, t) = m(H, 0) \cdot (1 + \varepsilon)^t$.
That is an above-average schema receives an exponentially increasing number of strings in the next generations.
The probability that single-point crossover disrupts a schema is precisely the probability that the crossover point falls within the schema's defining positions (those outermost, fixed bit-positions used to calculate the defining length). The probability that H survives crossover is greater than or equal to the term $1 - p_c \times \delta(H) / (l - 1)$. This survival probability is an inequality, because a disrupted schema might regain its composition if it crosses with a similar schema. The probability that H survives mutation is $(1 - p_m)^{o(H)}$, which can be approximated as $1 - o(H)p_m$ for small p_m and small $o(H)$. The product of the expected number of selections and the survival probabilities (with the smallest multiplicative term omitted) yields what is known as the *schema theorem*:

$$m(H, t+1) \geq m(H, t) \cdot \frac{f(H)}{\langle f \rangle} \cdot \left(1 - p_c \frac{\delta(H)}{l - 1} - o(H)p_m \right)$$

This equation states that short, low-order, above-average schemata receive exponentially increasing trials in subsequent generations of a genetic algorithm.

1.13 Parallel genetic algorithms

Part of the biological metaphor used to motivate genetic search is that GAs are inherently parallel. Beginning in 1987, a wide variety of parallel implementations have appeared in the literature. The most popular parallelizations of the canonical genetic algorithm are the coarse-grained genetic algorithm (*Island* model) and the fine-grained genetic algorithm (*Cellular* model).

- One motivation for using *Island* models is to exploit the coarse grain parallel model. Assume we wish to use 16 processors and have a population of 1600 strings. One way to evolve these strings is to break the total population down into subpopulations of 100 strings each. Each one of these subpopulations could then be evolved using a canonical genetic algorithm. Occasionally, the subpopulations would swap a few strings. This *migration* allows subpopulations to share genetic material [13].

Assume for a moment that one executes 16 separate genetic algorithms, each using a population of 100 strings *without migration*. In this case, 16 independent searches occur. This technique is sometimes called *partitioned* GA. The partitioned GA is highly redundant. This redundancy causes repeated exploration, from run to run, of certain regions of the search space (independent runs will blindly process the same subsolutions, since no coordination will

exist among processes). By introducing migration, the Island model is able to exploit differences in the various subpopulations. This variation, in fact, represents a source of genetic diversity. Each subpopulation is an island, and there is some designated way in which genetic structures are moved from one island to another. If a large number of strings migrate each generation, then global mixing occurs and local differences between islands will be driven out. If migration is too infrequent, it may not be enough to prevent each small subpopulation from prematurely converging. Tanese defines the *migration rate* as the percentage of each subpopulation that is exchanged. She (and others) finds that a migration rate has to be low (between 1 and 5%) in order to maintain stable, but differing subpopulations.
A coarse-grained genetic algorithm can be described as follows

```
generate multiple random subpopulations
generation number := 0
WHILE NOT termination condition DO
  FOR each subpopulation DO in parallel
      evaluate the individuals
      IF generation number MOD frequency = 0 THEN
        send R best individuals to a neighboring island
        receive R individuals from a neighboring island
        replace R individuals in the subpopulation
      ENDIF
      select individuals
      produce offspring
  ENDFOR
  generation number := generation number + 1
ENDWHILE
```

The variable R refers to the migration rate and and the variable *frequency,* expressed in number of generations, specifies the *exchange frequency.* Every *frequency* generations, an exchange takes place between each subpopulation and one of its neighbors.

- The cellular model assumes one individual resides at each processor. Each processor can pick the best string in its local neighborhood to mate with, or alternatively, some form of local probabilistic selection could be used. In either case, only one offspring is produced and becomes the new resident at that processor.
Muhlenbein‘s parallel, distributed GA (PGA) [9] places individuals on a two-dimensional grid, one individual per grid element or node. PGA works by hill-climbing to local maxima, then hopping to others via crossover. Each individual does the selection by itself. It looks for a partner in its neighborhood only, then performs crossover. If the single resulting offspring is fitter than its parent, it replaces the parent; otherwise the parent remains. The entire cycle then repeats. After few generations, there emerge many small subpopulations of similar strings with similar fitness values. These subpopulations are just as separate islands. This kind of separation is referred to as *isolation by distance*.
Manderick & Spiessens [8] proposed another fine-grained genetic algorithm which also assigns one individual to each processor. In this algorithm, the individuals of the population are placed on a planar grid and selection and mating are restricted to small neighborhoods on

that grid. Each processor crosses its element with an element selected from the processor’s neighborhood. Tournament selection, with the tournament size is given by the neighborhood size, is one possible selection method. The best of the two offspring produced by crossover replaces the original individual residing at that node.

A fine-grained genetic algorithm can be described as follows

```
FOR each grid point DO in parallel
     generate a random individual
ENDFOR
WHILE NOT termination condition DO
     FOR each grid point i DO in parallel
          evaluate individual in i
          select a neighboring individual j
          produce offspring from i and j
          assign one of the offspring to i
     ENDFOR
ENDWHILE
```

1.14 Comparison with other techniques

A number of other general purpose techniques have been proposed for use in optimization problems. Like GAs, they all assume that the problem is defined by a fitness function which must be maximized (or minimized).
There are a great many optimization techniques, some of the more general techniques are described below.

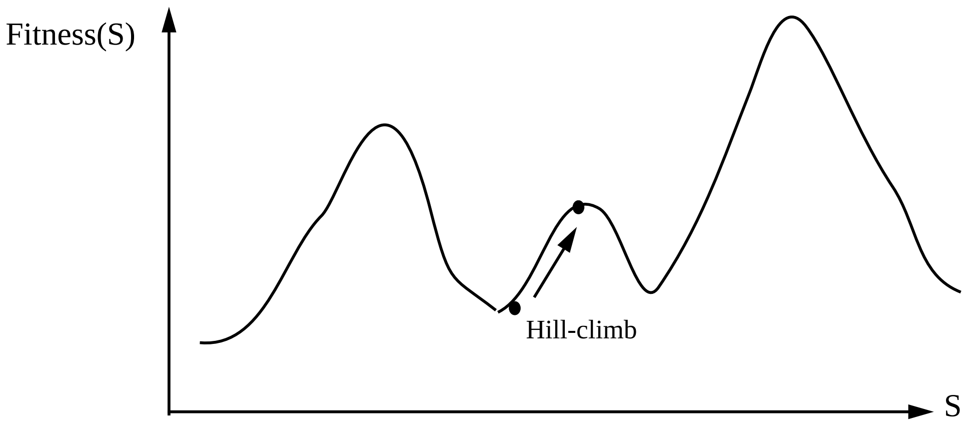
1.14.1 Random search

The brute force approach for difficult optimization problems is a random, or an enumerated search. Points in the search space are selected randomly, or in some systematic way, and their fitness evaluated. This is a very unintelligent strategy, and is rarely used by itself.

1.14.2 Hill-climbing

The hill-climbing method starts with a random configuration (point in the search space) and tries to improve it.
The improvement is carried out in small steps consisting of moving from one point to another. A move is selected randomly, the change in fitness value is computed, and if the change is positive the move is accepted and a new configuration is generated. Otherwise, the old configuration is kept. This process is repeated until there are no changes to the configuration that will increase the fitness function further. Below a version of the hill-climbing algorithm.

```
Generate a random initial solution $S_0$ (S := $S_0$)
Repeat
- Compute at random a neighbouring solution S'
- if Fitness(S') > Fitness(S) then S := S'
Until there is no better neighbour.
```



This method suffers from the problem that the first peak found will be climbed, and this may not be the highest peak. Having reached the top of a local maximum, no further progress can be made.
Another variation of this method is to combine the random search and hill-climbing. Once one peak has been located, the hill-climb is started again, but with another, randomly chosen, starting point. This technique (iterated search) has the advantage of simplicity and can perform well if the function does not have too many local maxima.

1.14.3 Simulated annealing

This technique, invented by Kirkpatrick [7] , is essentially a modified version of hill-climbing. The simulated annealing algorithm starts with choosing an initial configuration at random and calculates its fitness F. Then generates a new state and calculates its fitness F'. If the new configuration is higher in fitness than the old one $(F' - F = \Delta F > 0)$, this configuration is selected for the next step. If, however, the fitness was decreased the new configuration is not discarded but accepted with a certain parameter-controlled probability. This control parameter is commonly called temperature (T), making the thermodynamical origin of simulated annealing. The higher the temperature the higher the probability that configurations which decrease the fitness will be accepted.

$$P_{accept}(\Delta F, T) = 1 \qquad if\ \Delta F > 0$$

$$P_{accept}(\Delta F, T) = e^{\Delta F / T} \qquad if\ \Delta F \le 0$$

The main steps of the algorithm are given below.

```
1. Initialization step
   - Start with a random initial configuration ($S := S_0$)
   - $T := T_{max}$ (say 100)

2. Stochastic hill-climb
   - Generate and compute a random neighbouring state S'
     $\Delta F$ := Fitness(S') - Fitness(S)
   - Select the new configuration (S:=S') with probability $P_{accept}(\Delta F, T)$
   - Repeat this step until a number of configurations (typically 10) have been
     accepted or until a maximum number of iterations (typically 100) is
     exceeded

3. Convergence test
   - Set T := aT with 0 <= a < 1
   - If T >= $T_{min}$ (say 0.1) goto step 2
```

During the optimization process the control parameter is lowered, finally ending up with zero temperature, where only configurations increasing the fitness will be accepted. Note that negative moves are essential sometimes if local maxima are to be escaped.

- Like random search and hill-climbing, simulated annealing only deals with one candidate solution at a time, and so does not build up an overall picture of the search space. No information is saved from previous moves to guide the selection of new moves.
A GA, by comparison, starts with an initial random population, and allocates increasing trials to regions of the search space found to have high fitness. However, simulated annealing is still the topic of active research and has been used successfully in many applications.

Bibliography

[1] Baker, J. 1985. Adaptative Selection Methods for Genetic Algorithms. *Proc. 1st ICGA*, June 1985.

[2] Davis, L. 1991. Handbook of Genetic Algorithms. Van Nostrand Reinhold, 1991.

[3] De Jong, K. 1975. The Analysis and behaviour of a Class of Genetic Adaptive Systems. PhD thesis, University of Michigan, 1975.

[4] Goldberg, D.E. 1989. Genetic Algorithms in search, optimization and machine learning. Addison-Wesley, 1989.

[5] Holland, J.H. 1975. Adaptation in Natural and Artificial Systems. University of Michigan Press, 1975.

[6] Holland, J.H. 1992. Adaptation in Natural and Artificial Systems. MIT Press, 1992.

[7] Kirkpatrick, S., Gelatt C.D. and Vecchi M.P. 1983. Optimization by Simulated Annealing. Science, 220, 671 (1983).

[8] Manderick, B. and Spiessens P. 1989. Fine Grained Parallel Genetic Algorithms. *Proc. 3rd ICGA*, June 1989.

[9] Muhlenbein, H. 1991. Evolution in Time and Space - The Parallel Genetic Algorithm. Foundations of genetic algorithms, G. Rawlins, ed. Morgan-Kaufmann.

[10] Michalewicz, Z. 1992. Genetic Algorithms + Data Structures = Evolution Programs. Springer-Verlag, 1992.

[11] Oussaidène, M. and Chopard B. 1994. Optimisation des expressions arithmétiques sur une machine massivement parallèle en utilisant un algorithme génétique, SIPAR-Workshop on “Parallel and Distributed Computing”, university of Fribourg, October 1994.

[12] Syswerda, G. 1989. Uniform Crossover in Genetic Algorithms. *Proc. 3rd ICGA*, June 1989.

[13] Tanese, R. 1989. Distributed Genetic Algorithms. *Proc. 3rd ICGA*, June 1989.

[14] Whitley, D. 1989. The GENITOR Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials Is Best. *Proc. 3rd ICGA*, June 1989.

[15] Whitley, D. 1993. A Genetic Algorithm Tutorial. Technical Report CS-93-103, Colorado State University, November 1993.

2. Description of Genetic Programming

2.1 Introduction

The concept of artificial evolutionary process was introduced by Holland [3] and others during the sixties. He described a methodology for studying natural adaptative systems and designing artificial adaptative systems known as classifiers. Classifier systems are general, rule-based learning systems. The classifier rules are represented by sequences (strings) of symbols chosen from some (usually binary) alphabet. The searching of this representation space is performed using Genetic Algorithms (GAs).

As described in chapter 1, the principle of a GA consists of three main phases: evaluation, selection and reproduction. Each solution, in the "population", is evaluated to give some measure of its "fitness" (quality). Then, the more fit solutions are selected to be used as parents for the next iteration (generation). Those parents undergo genetic transformations (mutation and crossover) to form new solutions. This process is repeated some number of generations - the best solution hopefully represents the optimum solution or at least a sufficiently good one.

Koza [4] extended this genetic model of learning into the space of programs and thus introduced the concept of genetic programming. Each solution, in the search space, to the problem to solve is represented by a genetic program (GP), traditionally using the Lisp syntax.

A GP can be regarded as a Lisp function. It is usually represented by a parse tree. The number of nodes (including the terminal nodes) in the parse tree gives a measure of the complexity of that GP. Thus, the GP (* (+ 2 4) (- 6 1)) is of complexity 7. The terminal (non-terminal) nodes are randomly taken from some well defined TerminalSet (FunctionSet). In the creation phase, each parse tree is randomly built in recursive way, starting from the root node. To ensure the syntax validity and control the complexity expansion of GPs during this creation process, some rules must be observed. If the current node in the parse tree is a function taking r arguments then choose r nodes, from the Terminal/FunctionSet, to be child nodes. In the other case, the current node is a terminal, return to the parent node. A maximum depth is fixed before the execution so that when a branch in the parse tree reaches this level then a terminal must be chosen.

The evaluation phase consists of assigning a fitness value to each GP in the population. This fitness calculation usually requires evaluating each GP over a set of fitness cases. Each fitness case represents a particular situation in the search space. In image compression, the fitness cases are 2D array pixels. In a classification problem, the number of fitness cases corresponds to the amount of data to classify. In designing logical circuit taking k input bits, there are 2^k possible cases. In evolving trading strategies, the benchmark application described in chapter 5, each GP program is evaluated on a price time series and each series element represents a fitness case. The fitness value is used at selection phase so that more fit individuals will have more chance to be chosen. The selected individuals undergo genetic operations via mutation and crossover.

Genetic programming is now widely recognized as an effective search paradigm in artificial intelligence [1], databases [6], robotics [5] and many other areas .

2.2 Overview of Genetic Programming

Genetic programming overcomes the representation problem in genetic algorithms by increasing the complexity of the structures undergoing adaptation.
More precisely, the structures undergoing adaptation in genetic programming are hierarchical computer programs of dynamically varying size and shape. Different problems in artificial intelligence can be viewed as requiring discovery of a computer program producing some desired output for particular inputs. The process of solving these problems can be reformulated as a search for a highly fit individual computer program in the space of possible computer programs. In particular, the search space is the space of all possible computer programs composed of functions and terminals appropriate to the problem domain. Genetic programming provides a way to search for this fittest computer program.

In genetic pogramming, populations of computer programs are genetically bred. This breeding is perfomed using the *Darwinian* principle of survival of the fittest along with a genetic recombination operation appropriate for mating computer programs. A computer program that solves (or approximately solves) a given problem may emerge from this combination of natural selection and genetic operations.

Genetic programming starts with an initial population of randomly generated computer programs composed of functions and terminals appropriate to the problem domain. The functions may be standard arithmetic operations, mathematical functions, logical functions or domain-specific functions. The creation of this initial random population is a blind random search of the search space of the problem.

Each individual computer program in the population is measured in terms of how well it performs in the particular problem environment. This measure is called the fitness measure. The nature of the fitness measure varies with the problem. For many problems, fitness is naturally measured by the error produced by the computer program. The closer this error is to zero, the better the computer program.

Typically, each computer program in the population is run over a number of different fitness cases so that its fitness is measured as a sum or an average over a variety of representative different situations. These fitness cases sometimes represent a sampling of different values of an independent variable. For example, the fitness of an individual computer program in the population may be measured in terms of output produced by the program and the correct answer to the problem. This sum may be taken over a sampling of a number (say 50) different inputs to the program. The 50 fitness cases may be chosen at random or may be structured in some way. Unless the problem is so small and simple that it can be easily solved by blind random search, the computer programs in the initial population will have exceedingly poor fitness. Nonetheless, some individuals in the population will turn out to be somewhat fitter than others. These differences in the performance are then exploited. The Darwinian principle of survival of the fittest and the genetic operation (crossover) are used to create a new offspring population of individual computer programs from the current population of programs.

The genetic process of reproduction between two parental computer programs is used to create new offspring computer programs from two parental programs selected in proportion to fitness. The parental programs are typically of different sizes and shapes. The new offspring programs are composed of subexpressions (subtrees, subprograms, subroutines, building blocks) from their parents. These offspring programs are typically of different sizes and shapes than their parents.

Intuitively, if two computer programs are somewhat effective in solving a problem, then some of their parts probably have some merit. By recombining randomly chosen parts of somewhat effective programs, we may produce new computer programs that are even fitter in solving the problem.
After the genetic operations are performed on the current population, the population of offspring (i.e., the new generation) replaces the old population (i.e., the old generation).
Each individual in the population of computer programs in then measured for fitness and the process is repeated over many generations.
This algorithm will produce populations of computer programs which, over many generations, tend to exhibit increasing average fitness in dealing with their environment.
The best individual that appeared in any generation of a run (the best-so-far individual) is designated as the result produced by genetic programming.

The hierarchical character of the computer programs that are produced is an important feature of genetic programming. The results of genetic programming are inherently hierarchical. In many cases, the results produced by genetic programming are default hierarchies, prioritized hierarchies of tasks, or hierarchies in which one behavior subsumes or suppresses another.
The dynamic variability of the computer programs that developed along the way to a solution is also an important feature of genetic programming. It would be difficult and unnatural to try to specify or restrict the size and shape to the eventual solution in advance. Moreover, advance specification or restriction of the size and shape of the solution to a problem narrows the window by which the system views the search space and might well preclude finding the solution to the problem.

Another important feature of genetic programming is that the inputs, intermediate results, and outputs are typically expressed directly in terms of the natural terminology of the problem domain. The computer programs produced by genetic programming consist of functions that are natural for the problem domain. Finally, the structures undergoing adaptation in genetic programming are active. They are not passive encodings of the solution to the problem. Instead, given a computer on which to run, the structures in genetic programming are active structures that are capable of being executed in their current form.
The genetic programming paradigm is a domain-independent method. It provides a unified approach to the problem of finding a computer program to solve a problem. A wide variety of seemingly different problems can be reformulated into a common form: induction of a computer program.

In summary, the genetic programming paradigm breeds computer programs to solve problems by executing the following steps:

1. Generate an initial population of random compositions of the functions and terminals of the problem (computer programs).

2. Iteratively perform the following substeps until the termination criterion has been satisfied:

a. Execute each program in the population and assign it a fitness value according to how well it solves the problem.

b. Create a new population of computer programs by applying the following two primary operations. The operations are applied to computer program(s) in the population chosen with a probability based on fitness.

- Create a new computer program by genetically mutating a randomly chosen function (or terminal) of an existing individual.

- Create new computer programs by genetically recombining randomly chosen parts of two existing programs.

3. The best computer program that appeared in any generation (the best-so-far individual) is designated as the result of genetic programming. This result may be a solution (or an approximate solution) to the problem.

Figure 6 shows a flowchart of these steps for the genetic programming paradigm. The parameters P_c and P_m control respectively the frequencies of crossover and mutation. The index j refers to an individual in a population of size P. The variable G indicates the current generation number.

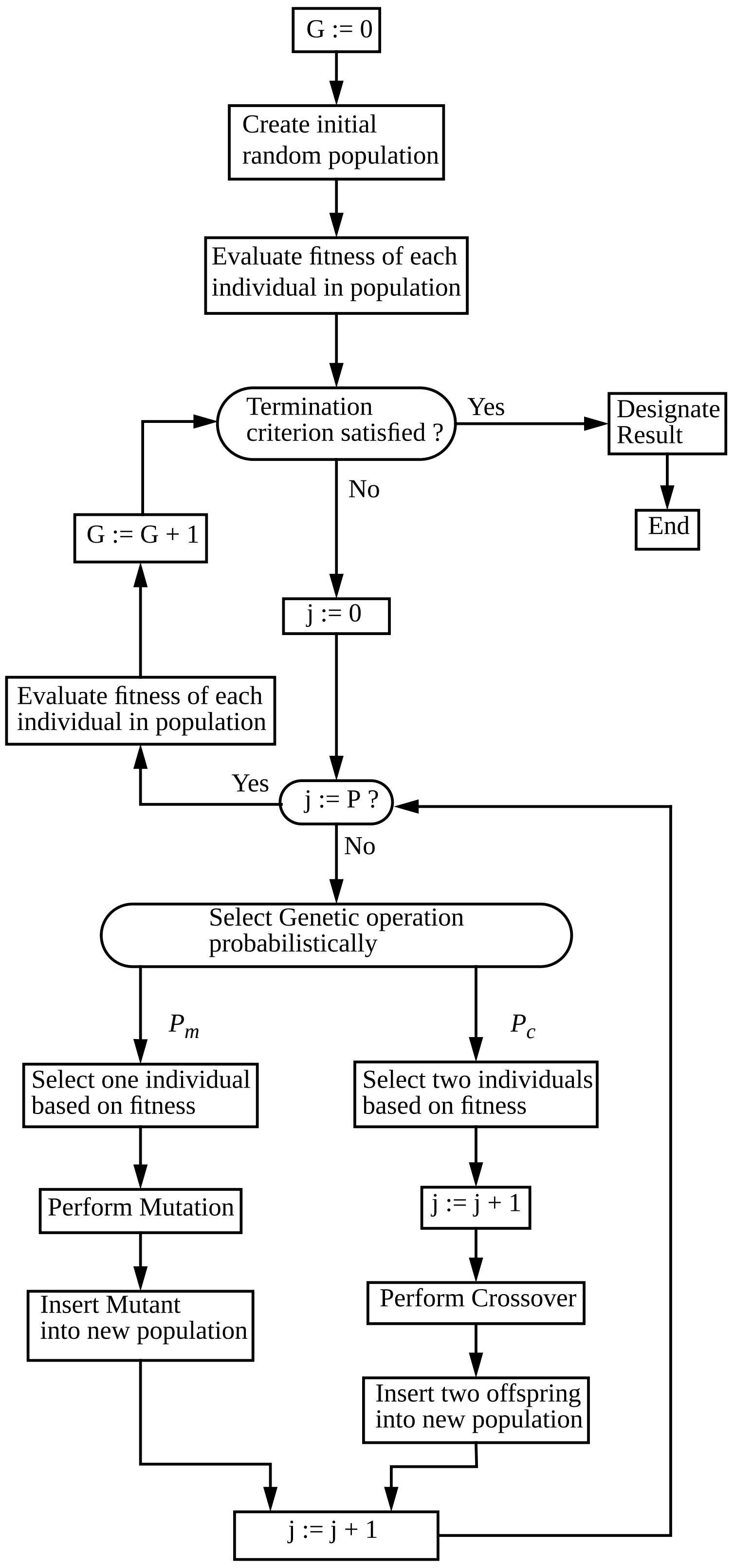

G := 0
Create initial random population
Evaluate fitness of each individual in population
Termination criterion satisfied ?
Yes
Designate Result
End
No
G := G + 1
j := 0
Evaluate fitness of each individual in population
Yes
j := P ?
No
Select Genetic operation probabilistically
P_m
P_c
Select one individual based on fitness
Select two individuals based on fitness
Perform Mutation
j := j + 1
Perform Crossover
Insert Mutant into new population
Insert two offspring into new population
j := j + 1

fig 6: Flowchart for the genetic programming paradigm.

2.3 The structures undergoing adaptation

For the conventional genetic algorithm and genetic programming, the structures undergoing adaptation are a population of individual points from the search space, rather than a single point. Genetic methods differ from most other search techniques in that they simultaneously involve a parallel search involving hundreds or thousands of points in the search space.

The individual structures that undergo adaptation in genetic programming are hierarchically structured computer programs. The size, the shape, and the contents of these computer programs can dynamically change during the process.

The set of possible structures in genetic programming is the set of all possible compositions of functions that can be created recursively from the set of N_f functions

$F = \{f_1, f_2, \ldots, f_{N_f}\}$ and the set of N_t terminals from $T = \{a_1, a_2, \ldots, a_{N_t}\}$.

Each particular function f_i in the *FunctionSet F* takes a specified number r_i of arguments. That is function f_i has arity r_i .

The functions in the *FunctionSet* may include

- arithmetic operations (+, -, *, etc.),
- mathematical functions (such as sin, cos, exp, and log),
- Boolean operations (such as AND, OR, NOT),
- conditional operators (such as If-Then-Else),
- function causing iteration (such as Do-Until), and
- any other domain-specific functions that may be defined.

The terminals are typically either variable atoms (like *X*, *N*, representing, perhaps, the inputs, detectors, or state variables of some system) or constant atoms (such as the number 3 or the Boolean constant TRUE). Occasionally, the terminals are functions taking no explicit arguments, the real functionality of such functions lying in their side effects on the state of the system.

Consider the *FunctionSet*
F = { + , - , / , * }
and the *TerminalSet*
T = { X , N }
where X and N are numerical variable atoms that serve as arguments for the functions.
X is real variable atom and N is the set of natural numbers.

We can combine the set of functions and terminals into a combined set C as follows:

C = $F \cup T$ = { + , - , / , * , X , N } .

We can then view the terminals in the combined set C as functions requiring zero arguments in order to be evaluated. That is, the six items in the set C can be viewed as taking 2, 2, 2, 2, 0 and 0 arguments, respectively.

For example, consider the logistic function *4x (1-x)*.
This real-valued function can be expressed by the following LISP S-expression:
((* 4 x) (- 1 x))* .

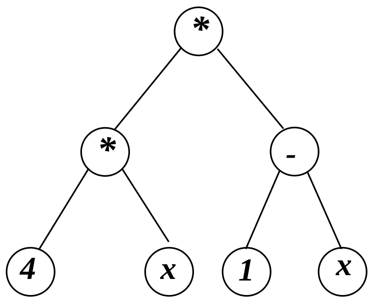


fig 7: Graphical representation of the S-expression ((* 4 x) (- 1 x))*

Figure 7 graphically depicts the above LISP S-expression as a rooted, point-labeled tree with ordered branches. The three function nodes of the tree are labeled with functions (* , * , -). The four terminal nodes (leaves) of the tree are labeled with terminals 4 , x , 1 and x respectively. The root of the tree is labeled with the function appearing just inside the outermost left parenthesis of the LISP S-expression (the *).

The search space for genetic programming is the space of all possible LISP S-expressions that can be recursively created by compositions of the available functions and available terminals for the problem.

The structures that undergo adaptation in genetic programming are hierarchical structures. The structures that undergo adaptation in the conventional genetic algorithm are one-dimensional fixed-length linear strings.

In genetic programming, the *TerminalSet* and the *FunctionSet* should be selected so as to satisfy the requirements of closure and sufficiency. These properties are defined below.

2.4 Closure of the *FunctionSet* and *TerminalSet*

The closure property requires that each of the functions in the function set be able to accept, as its arguments, any value and data type that may possibly be returned by any function in the *FunctionSet* and value and data type that may possibly be assumed by any terminal in the

TerminalSet. That is, each function in the *FunctionSet* should be well defined and closed for any combination of arguments that it may encounter.

In ordinary programs, arithmetic operations operating on numerical variables are sometimes undefined (e.g., division by zero). Many common mathematical functions operating on numerical variables are also sometimes undefined (e.g., logarithm of zero). In addition, the value returned by many common mathematical functions operating on numerical variables is sometimes a data type that is unacceptable in a particular program (e.g., square root or logarithm of a negative number).

Closure can be achieved in a straightforward way for the vast majority of problems merely by careful handling of a small number of situations.

If the arithmetic operation of division can encounter the numerical value of 0 as its second argument, the closure property will not be satisfied unless some arrangement is made to deal with the possibility of division by 0. One simple approach to guarantee closure is to define a protected division function. The protected division function DIV takes two arguments and returns one when division by 0 is attempted (including 0 divided by 0), and, otherwise, returns the normal quotient. It might be programmed as follows in *C* :

```
FITNESS DIV( FITNESS numerator denominator )
{
     if ( denominator == 0 ) return 1;
     else return (numerator / denominator);
}
```

If the square root function can encounter a negative argument or if the logarithm function can encounter a nonpositive argument in a problem where the complex number that ordinarily would be returned is unacceptable, we can guarantee closure by using a protected function.
For example, the protected square root function SRT takes one argument and returns the square root of the absolute value of its argument. It might be programmed as

```
FITNESS SRT ( FITNESS argument)
{
     return sqrt(abs(argument));
}
```

where SQRT is the common *C* square root function.

The protected natural logarithm function RLOG returns 0 if its one argument is 0 and otherwise returns the natural logarithm of the absolute value of its argument. It might be programmed as

```
FITNESS RLOG ( FITNESS argument)
{
     if (argument == 0) return 0;
     else return log(abs(argument));
}
```

where LOG is the common *C* natural logarithm function.

If a program contains a conditional operator in a problem where the Boolean value that would ordinarily be returned is unacceptable, then the conditional operator can be modified so as to return numerical values.

If numerical-valued logic is used, a numerical-valued conditional comparative operator is defined so as to return numbers (such as + 1 and - 1 or 1 and 0) instead of returning Boolean values (i.e., TRUE and FALSE).
For example, the numerical-valued greater than function GT over two arguments would be defined so as to return +1 if its first argument is greater than its second argument and to return -1 otherwise. Such a function does not introduce a Boolean value into the program. The numerical-valued greater-than function GT might be programmed as

```
FITNESS GT(FITNESS first_argument second_argument)
{
     if (first_argument > second_argument) return 1;
     else return -1;
}
```

A conditional comparative operator can be defined so as to first perform the desired comparison and to then execute an alternative depending on the outcome of the comparison test. For example, the conditional comparative operator IFLTZ (If Less Than Zero) can be defined over three arguments so as to execute its second argument if its first argument is less than 0, but to execute its third argument otherwise. Such an operator returns the result of evaluating whichever of the second and third arguments is actually selected on the basis of the outcome of the comparison test. It therefore does not introduce a Boolean value into the program.

The closure property is desirable, but is not absolutely required. If this closure property does not prevail, we must then address alternatives such as discarding individuals that do not evaluate to an acceptable result or assigning some penalty to such infeasible individuals.

Note that the closure property is required only for terminals and functions that may actually be encountered. If the structures undergoing adaptation are known to comply with the constraints required by the syntax of the rules of construction, closure is required only over the values of terminals and values returned by functions that will actually be encountered.

2.5 Sufficiency of the *FunctionSet* and the *TerminalSet*

The sufficiency property requires that the set of terminals and the set of primitive functions be capable of expressing a solution to the problem. Some composition of the supplied functions and terminals can yield a solution to the problem.

Depending on the problem, this identification step may be obvious or may require considerable insight.

For example, in the domain of Boolean functions, the *FunctionSet*
F = {AND , OR , NOT}
is known to be sufficient for realizing any Boolean function.

The choice of the set of available functions and terminals, of course, directly affects the character and appearance of the solutions. The available functions and terminals form the basis for generating potential solutions.

2.6 The Initial Population

The generation of each individual S-expression in the initial population is done by randomly generating a rooted, point-labeled tree with ordered branches representing the S-expression.

The process starts selecting one function from the set F at random (using a uniform probability distribution) to be the label for the root of the tree. The selection of the label for the root of the tree is restricted to the *FunctionSet* F in order to generate a hierarchical structure, thus avoiding a degenerate structure consisting of a single terminal.

Figure 8 shows the creation of the root of a random parse tree. The function + was selected from the *FunctionSet* F as the label for the root of the tree.

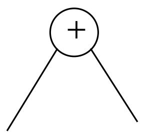


fig 8: Beginning of the creation of a random parse tree. The function + is chosen as the root of the tree.

Whenever a point of the tree is labeled with a function *f* from F, then *r* nodes, where *r* is the number of arguments taken by the function *f*, are created from that point. Then, for each such function node, an element from the combined set $C = F \cup T$ of functions and terminals is randomly selected to be the label for the endpoint of that function node.

If a function is chosen to be the label for any such endpoint, the generating process then continues recursively. For example, in figure 9, the function * (point 2) , from the combined set *C* , was selected as the label of the first argument of the function + (point 1). The function * takes two arguments.

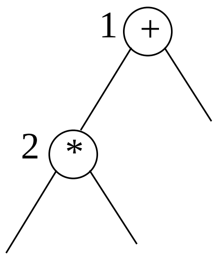


*fig 9: Continuation of the creation of a random parse tree. The function * is chosen for point 2.*

If a terminal is chosen to be the label for any point, that point becomes an endpoint of the tree and the generating process is terminated for that point.
For example, in figure 10, the terminal *A* from the *TerminalSet* T was selected to be the label of the first argument corresponding to the point labeled with the function *. Similarly, the terminals *B* and *C* were selected to be labels of the two other arguments. This process continues recursively from left to right until a completely labeled tree has been created.

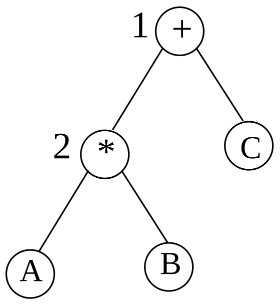


fig 10: Completion of the creation of a random parse tree. The terminals A, B, and C were chosen.

- This generative process can be implemented in several different ways resulting in initial random trees of different sizes and shapes.

Two of the basic ways are called *grow* method and the *variable* method. The depth of a tree is defined as the length of the longest nonbacktracking path from the root node to a terminal node.

1. The *grow* method of generating the initial random population involves creating trees for which the length of every nonbacktracking path between a terminal node and the root node is equal to the specified maximum depth.
This is accomplished by restricting the selection of the label for points at depths less than the maximum to the *FunctionSet* F, and then restricting the selection of the label for points at the maximum depth to the *TerminalSet* T.

2. The *variable* method of generating the initial random population involves growing trees that are variably shaped. The length of a path between a terminal node and the root node is no greater than the specified maximum depth.
This is accomplished by making the random selection of the label for points at depths less than the maximum from the combined set $C = F \cup T$ consisting of the union of the *FunctionSet* F and the *TerminalSet* T, while restricting the random selection of the label for points at the maximum depth to the *TerminalSet* T.

The most frequently used generative method is called *ramped half-and-half*. In genetic programming, the size and shape of the solution are usually not known in advance. The *ramped half-and-half* generative method produces a wide variety of trees of various sizes and shapes.

3. The *ramped half-and-half* generative method is a mixed method that incorporates both the *grow* method and the *variable* method.

The *ramped half-and-half* generative method involves creating an equal number of trees using a depth parameter that ranges between 2 and the maximum specified depth. For example, if the maximum specified depth is 6 (the default value), 20% of the trees will have depth 2, 20% will have depth 3, and so forth up to depth 6. Then, for each value of depth, 50% of the trees are created via the *grow* method and 50% of the trees are produced via the *variable* method.

Note that, for the trees created with the *grow* method for a given depth, all paths from the root of the tree to an endpoint are the same length and therefore have the same shape. In contrast, for the trees created via the *variable* method for a given value of depth, no path from the root of the tree to an endpoint has a depth greater than the given value of depth. Therefore, for a given value depth, these trees vary considerably in shape from one another.

Thus, the *ramped half-and-half* method creates trees having a wide variety of sizes and shapes.

Duplicate individuals in the initial random generation are unproductive deadwood; they waste computational resources and undesirably reduce the genetic diversity of the population. Thus, it is desirable, but not necessary, to avoid duplicates in the initial random population. In genetic programming, duplicate random individuals are especially likely to be created in the initial random generation when the trees are small. Thus, each newly created S-expression is checked for uniqueness before it is inserted into the initial population. If a new S-expression is a duplicate, the generating process is repeated until a unique S-expression is created.

The variety of a population is the percentage of individuals for which no exact duplicate exists elsewhere in the population. If duplicate checking is done, the variety of the initial random population is 100%. In later generations, the creation of duplicate individuals via genetic operations is an inherent part of genetic processes.

It should be remembered that inserting relatively high-fitness individuals into an initial population of random individuals will after one generation, result in almost total dominance of the population by copies and offspring of the primed individuals. In terms of genetic diversity, the result will be, after only one generation, very similar to starting with a population of size equal to the relatively tiny number of primed individuals.

2.7 Genetic Operations for Modifying Structures

This section describes the two primary operations used to modify the structures undergoing adaptation in genetic programming.

- Crossover (binary operator)
- Mutation (unary operator)

2.7.1 Crossover

The crossover (sexual recombination) operation for genetic programming creates variation in the population by producing new offspring that consist of parts taken from each parent. The crossover operation starts with two parental S-expressions and produces two offspring S-expressions. That is, it is a sexual operation.

The first parent is chosen from the population by a fitness-based selection method. The second parent is chosen by means of the same selection method (that is, with a probability equal to its normalized fitness). The different selection methods are described in section 1.9.

Once the parents have been chosen, the operation begins by independently selecting, using a uniform probability distribution, one random point in each parent to be the crossover point for that parent. Note that the two parents typically are of unequal size.

The crossover fragment for a particular parent is the rooted subtree which has as its root the crossover point for that parent and which consists of the entire subtree lying below the crossover point. This subtree sometimes consists of one terminal.

The first offspring S-expression is produced by deleting the crossover fragment of the first parent from the first parent and then inserting the crossover fragment of the second parent at the crossover point of the first parent. The second offspring is produced in a symmetric manner.

For example, consider the two parental LISP S-expressions in figure 11. The functions appearing in these two S-expressions are the arithmetic +, - and * functions. The terminals appearing in this figure are the numerical arguments *A* , *B* and *C*.

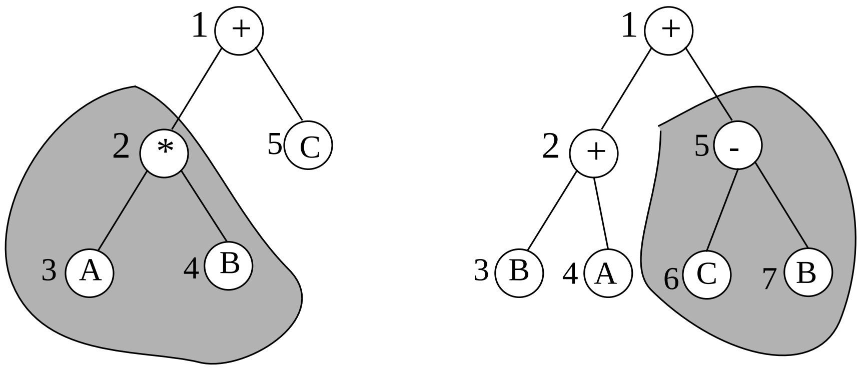


fig 11: Two parental genetic programs.

Equivalently, in terms of LISP S-expressions, the two parents are

*(+ **(* A B)** C)*

and

*(+ (+ B A) **(- C B)**) .*

Assume that the points of both trees above are numbered in a depth-first, left-to-right way. Suppose that the second point (out of the five points of the first parent) is randomly selected as the crossover point for the first parent. The crossover point of the first parent is therefore the * function. Suppose also that the fifth point (out of the seven points of the second parent) is selected as the crossover point of the second parent. The crossover point of the second parent is therefore the - function. The portions of the two parental S-expressions in boldface in figure 11 are the *crossover fragments*. The remaining portions of the two parental S-expressions in figure 11 are called the *remainders*.

Figure 12 depicts these two crossover fragments and figure 13 shows the two offspring resulting from crossover.

The first offspring S-expression in figure 13 is *(+ **(- C B)** C)*

and the second offspring is *(+ (+ B A) **(* A B)**) .*

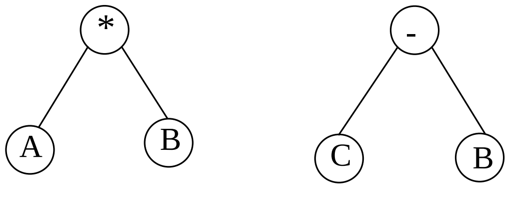


fig 12: The crossover fragments resulting from selection of point 2 of the first parent and point 5 of the second parent as crossover points.

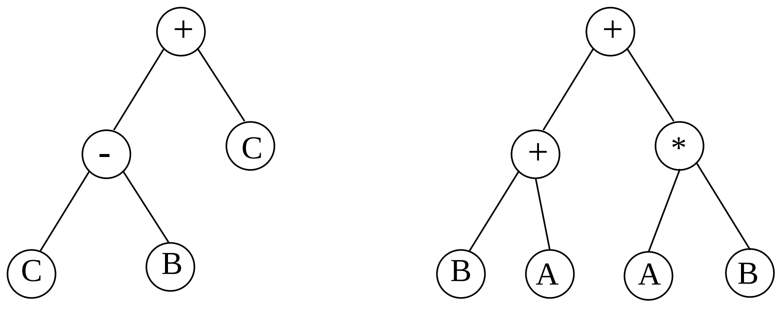


fig 13: The two offspring produced by crossover.

Because entire subtrees are swapped, and because of the closure property of the functions themselves, this genetic crossover (recombination) operation always produces legal syntax LISP S-expressions as offspring regardless of the selection of parents or crossover points.

If a terminal is located at the crossover point in precisely one part, then the subtree from the second parent is inserted at the location of the terminal in the first parent (thereby introducing a subtree instead of a single terminal point) and the terminal from the first parent is inserted at the location of the subtree in the second parent.

If terminals are located at both crossover points selected, the crossover operation merely swaps these terminals from tree to tree. The effect of crossover, in this case, is a point mutation. Thus, occasional point mutation is an inherent part of the crossover operation.

If the root of one parental S-expression happens to be selected as the crossover point, the crossover operation will insert the entire first parent into the second parent at the crossover point of the second parent. In this event, the entire first parent will become a subtree within the second parent. This will often have the effect of producing an offspring with considerable depth. In addition, the crossover fragment of the second parent will then become the other offspring.

In the rare situation where the root of one parental S-expression happens to be selected as the crossover point and the crossover fragment from the second parent happens to be a single terminal, then the first parent becomes a single terminal and the other offspring will be a LISP s-expression.

If the roots of two parents both happen to be chosen as crossover points, then the crossover operation simply copies those two parents.

When an individual mates with itself or when two identical individuals mate the two resulting offspring will generally be different (because the crossover points selected are, in general, different for the two parents). This is in contrast to the case of the conventional genetic algorithm operating on fixed-length character strings where the one selected crossover point applies to both parents.

In the conventional genetic algorithm, when an individual mates with itself (or copies of itself), the two resulting offspring will be identical. This fact fortifies the tendency toward convergence in the conventional genetic algorithm.

In contrast, in genetic programming, when an individual mates with itself (or copies of itself), the two resulting offspring will, in general, be different (except in the relatively infrequent case when the crossover points are the same). The crossover operation exerts a counterbalancing pressure away from convergence. A maximum permissible size (measured via the depth of the tree) is established for offspring created by the crossover operation. This limit prevents the expansion of large amounts of computer time on a few extremely large genetic programs.

A default value of 17 for this maximum permissible depth, permits potentially enormous programs. For example, the largest permissible LISP program consisting of entirely diadic functions would contain $2^{17} = 131072$ functions and terminals.

2.7.2 Mutation

The mutation operation introduces random changes in structures in the population. In conventional genetic algorithms operating on strings, the mutation operation can be beneficial in reintroducing diversity in a population that may be tending to converge prematurely.

In the conventional genetic algorithm, it is common for a particular symbol (i.e., an allele) appearing at a particular position on a chromosome string to disappear at an early stage of a run because that particular allele is associated with inferior performance, given the alleles prevailing at other positions of the chromosome string at that stage of the run.

The lost allele may be precisely what is needed to achieve optimal performance at a later stage of the run.

In this situation, the mutation operation may occasionally have beneficial results. Nonetheless, it is important to recognize that the mutation operation is a relatively unimportant secondary operation in the conventional genetic algorithm [3] [2].
Mutation is asexual and operates on only one parental S-expression. The individual is selected with a probability proportional to the normalized fitness. The result of this operation is one offspring S-expression.

The mutation operation begins by selecting a point at random within the S-expression. This mutation point can be an internal (i.e., function) point or an external (i.e., terminal) point of the tree. The mutation operation then removes whatever is currently at the selected point and whatever is below the selected point and inserts a randomly generated subtree at that point.

This operation is controlled by a parameter that specified the maximum size (measured by depth) for the newly created subtree that is to be inserted. This parameter typically has the same value as the parameter for the maximum initial size of S-expressions in the initial random population.

A special case of mutation operation involves inserting a single terminal at a randomly selected point of the tree. This point mutation occurs occasionally in the crossover operation when the two selected crossover points are both terminals.

For example, in the "before" diagram in figure 14, point 2 (i.e., *) of the S-expression was selected as the mutation point. The terminal *B* was randomly generated and inserted at that point to produce the S-expression shown in the "after" diagram.

In genetic programming, particular functions and terminals are not associated with fixed positions in a fixed structure. Moreover, when genetic programming is used, there are usually considerably fewer functions and terminals for a given problem than there are positions in the chromosome in the conventional genetic algorithm. Thus, it is relatively rare for a particular function or terminal ever to disappear entirely from a population in genetic programming. Therefore, in genetic programming, the mutation operator does not serve the potentially important role of restoring lost diversity in a population, as it does in the conventional genetic algorithm.

Note that, in genetic programming, whenever the two crossover points in the two parents happen to both be endpoints of trees, the crossover operates in a manner very similar to point mutation.

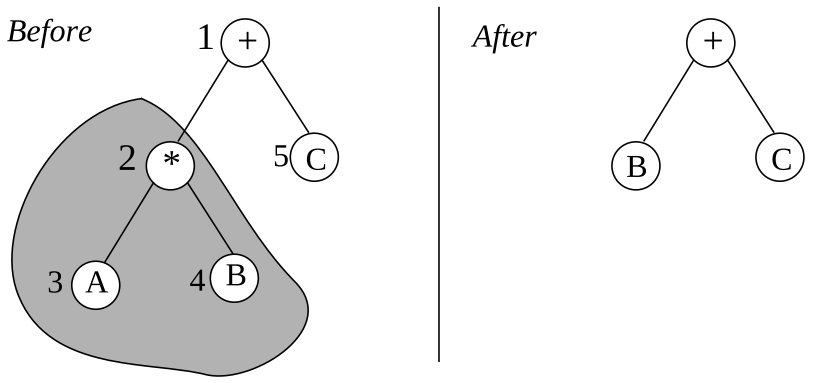


fig 14: A computer program before and after the mutation operation is performed at point 2.

2.8 State of the Adaptive System

In genetic programming, the state of the adaptive system at any point during the process consists only of the current population of individuals. No additional memory or centralized bookkeeping is necessary.

In a computer implementation of the genetic programming paradigm, it is also necessary to cache the control parameters for the run, the *TerminalSet* and the *FunctionSet* (if mutation is being used), and the best-so-far individual if it is being used as part of the process of result designation for the run.

2.9 Termination criterion

A run of the genetic programming paradigm terminates when the termination criterion is satisfied. The termination criterion for genetic programming is that the run terminates when either a prespecified maximum number *G* of generations have been run (the generational predicate) or some additional problem-specific success predicate has been satisfied.

The success predicate often involves finding a 100%-correct solution to the problem. For problems where a solution may not be easily recognized (optimization problems) or problems where we do not ever expect an exact solution (creating mathematical models for noisy empirical data), some appropriate lower criterion for success is usually adopted for purposes of terminating a run. For some problems, there is no success predicate; the results are analyzed after running for *G* generations.

2.10 Result Designation

The method of result designation for genetic programming is to designate the best individual that ever appeared in any generation of the population (i.e., the *best-so-far* individual) as the result of a run of genetic programming.
Note that the *best-so-far* individual is kept separately and not inserted in all subsequent generations (i.e., the so-called *elitist* strategy is not followed). The *best-so-far* individual is merely cached and reported as the result of the entire run when the run eventually terminates according to the termination criterion.
When this method of result designation is used, the state of the system consists of the current population of individuals and the one cached *best-so-far* individual.

2.11 Default Parameters

Control parameters characterizing runs of genetic programming are either numerical or qualitative. The values of these parameters are fixed at the default values used in the vast majority of cases. The two major numerical parameters are the population size, *P*, and the maximum number of generations to be run, *G*.

- The default population size, *P*, is 4000.

- The default value for the maximum number of generations to be run, *G*, is 51 (an initial random generation, called generation 0, plus 50 subsequent generations).

- The probability of crossover, p_c, is 0,90. That is, crossover is performed such that the number of individuals produced as offspring by the crossover operation is equal to 90% of the population size on each generation. For example, if the population size is 16000, then 14400 individuals are produced as offspring by the crossover operation on each generation.

- In choosing crossover points, we use a probability distribution that allocates p_f = 90% of the crossover points equally among the function nodes of each parse tree and $p_t = 1 - p_f$ = 10% of the crossover points equally among the terminal nodes of each parse tree.

- A maximum size (measured by depth), $D_{crossover}$, is 17 for genetic programs created by the crossover operation.

- A maximum size (measured), $D_{creation}$, is 6 for the random individuals generated for the initial population.

- The probability of mutation, p_m, specifying the frequency of performing the operation of mutation is 0.

- The generative method for the initial random population is ramped half-and-half.

- The method of selection for the first parent in crossover is tournament selection (with a group size of seven).

- The method of selecting the second parent for a crossover is the same as the method for selecting the first parent (i.e., tournament selection with a group size of seven).

- The elitist strategy is not used.

Bibliography

[1] Bennett, F. H. 1996. Automatic creation of an Efficient Multi-Agent Architecture Using genetic programming with Architecture-Altering Operations. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996*, pages 30-38. MIT Press, 1996. Proceedings of the first Annual Conference, July 28-31, 1996, Stanford University.

[2] Goldberg, D.E. 1989. Genetic Algorithms in search, optimization and machine learning. Addison-Wesley, 1989.

[3] Holland, J. 1975. Adaptation in Natural and Artificial Systems, University of Michigan Press, Ann Arbor.

[4] Koza, J. 1992. Genetic programming, MIT Press.

[5] Ross, S. J., Daida J. M., Doan C. M., Bersano-Begey T. F. and McClain J. J. 1996. Variations in Evolution of Subsumption Architectures Using Genetic Programming: The Wall Following Robot Revisited. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996*, pages 191-199. MIT Press, 1996. Proceedings of the first Annual Conference, July 28-31, 1996, Stanford University.

[6] Sitllger, M. and Spiliopoulou M. 1996. Genetic programming in Database Query Optimization. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996*, pages 388-393. MIT Press, 1996. Proceedings of the first Annual Conference, July 28-31, 1996, Stanford University.

3. A Parallel Genetic Programming System

3.1 Introduction

This chapter describes a parallel implementation of the genetic programming paradigm on distributed memory machines. The code is implemented on the IBM SP-2 machine and written in C++, using the PVM3 message passing library [4]; it can be easily ported on other parallel machines like the Cray T3D and on workstation clusters. The sequential version of the code is based on [3].

- The Parallel Genetic Programming System (*PGPS*) uses subtree crossover. Crossover operator selects two GPs from the population and chooses one node (crosspoint) on each. Each node is, by definition, the root of some complete subtree. The two subtrees are extracted and each is swapped with the other. Note that the syntax of the resulting GPs is valid and only the maximum depth constraint is checked for non-violation.

- To mutate a genetic program, PGPS uses the point mutation to modify any function or terminal node. In point mutation, any terminal node can be replaced by any other terminal taken from the *TerminalSet*, but a function node can only be replaced by another function (from the *FunctionSet*) which has the same number of arguments.
Note that the mutation genetic operator is unary. Also, the point mutation does not modify the complexity of the GP.

- The selection method used in PGPS is the tournament selection (described in section 1.9).

We present a parallel scheme for genetic programming which maintains multiple independent subpopulations (also known as islands) interacting asynchronously using a ring topology. The evaluation phase in each evolutionary process is parallelized and separated from the rest of the population management calculations.

3.2 PGPS scheme

The inherent convergence characterizing traditional GAs makes that different high fitness individuals can’t be maintained in single population: once a suboptimal individual dominates the population, selection is likely to keep it and prevent further adaptation. Evolving multiple, independent subpopulations with occasional interchange (migration) of solutions between these subpopulations is an alternative approach to deal with this premature convergence problem. This allows not only a better exploration of the global search space (each subpopulation explores different parts of the search space and maintains its own high fitness individuals), but also retards premature convergence by introducing, via migration, diversity from other subpopulations.

The parallel genetic programming system maintains multiple independent subpopulations (evolutionary processes) interacting asynchronously using the ring topology.

Due to large number of fitness cases in complex applications, like time series modelling, the evaluation phase takes most of the run-time. Particularly, in evolving financial trading strategies [7], each GP is evaluated over a price time series exceeding 24×10^4 elements; thus, the time spent in the selection and reproduction phases is practically negligible compared with the population evaluation time.

Parallelizing the different evaluation phases related to each evolutionary process in the subpopulations, on a parallel coarse-grain machine such as the IBM SP-2, can be done naturally. After the reproduction phase, each GP is simply sent to a processing node for evaluation, independently of operations in other processing nodes [8]. However, the run time GP complexity with programs of widely differing sizes inside each subpopulation unbalances the work load on the allocated nodes and thus making the design of a parallel algorithm and its implementation to obtain large speedups a nontrivial task. This irregularity causes some processing nodes to be idle while others are active. In fact, this problem is equivalent to finding an optimal schedule of p independent tasks on $m < p$ machines and is known to be NP-hard.

Section 4.2 analyses load balancing and investigates different strategies for the computational load distribution. We first consider a static scheduling algorithm to distribute the genetic programs upon the processing nodes at the evaluation phase. Next, we improve the processor utilization by a dynamic load balancing algorithm based on run time GP complexity.
The evaluation phase, in each evolutionary process, is separated from the rest of GA calculations. Interleaving the different evaluation phases allows to hide message latencies by switching between multiple threads. Our implementation shows that the parallelization of genetic programming on distributed memory machines is linearly scalable with respect to the number of processors [9].

PGPS consists of multiple master-slave instances each mapped on all the allocated processing nodes (see figure 15). Each master process implements the genetic programming management system. A slave process corresponds to the user defined problem. The number of master processes may vary from 1 to m , where m is the total number of available processors.

The conventional master-slave paradigm is used as model template where each instance evolves its own population. The subpopulation, maintained by each evolutionary process, is evaluated using all the allocated nodes. The genetic programming management system creates the initial population, applies the genetic operators (crossover and mutation) and performs the selection of genetic programs which will be the candidates to the reproduction phase. At the evaluation phase, each master process distributes the work load among all the processing nodes in the virtual machine, including the processor on which the current master process runs. The interprocessor communications are achieved using communication interface routines.

The genetic programming management system packs each parse tree from its memory representation into a buffer and sends it to the appropriate slave (or master) process using PVM routines. Each slave process receives the character string into a buffer and performs the unpack operation to build the equivalent parse tree in memory. This unpack operation requires the translation of the GP to generate an intermediate form suitable to memory representation. After the fitness calculation of a genetic program, each slave (or master) process sends back the calculated fitness value to the source master process. After work distribution, each master process first evaluates the individuals belonging to its own subpopulation (local tasks), next switches to the GPs coming from other subpopulations (external tasks). The local tasks require no communication. Note that GPs can be evaluated independently and therefore there is no communication between the slaves. The load distribution cost and the communication overhead are negligible compared with the computational cost associated with fitness calculation.

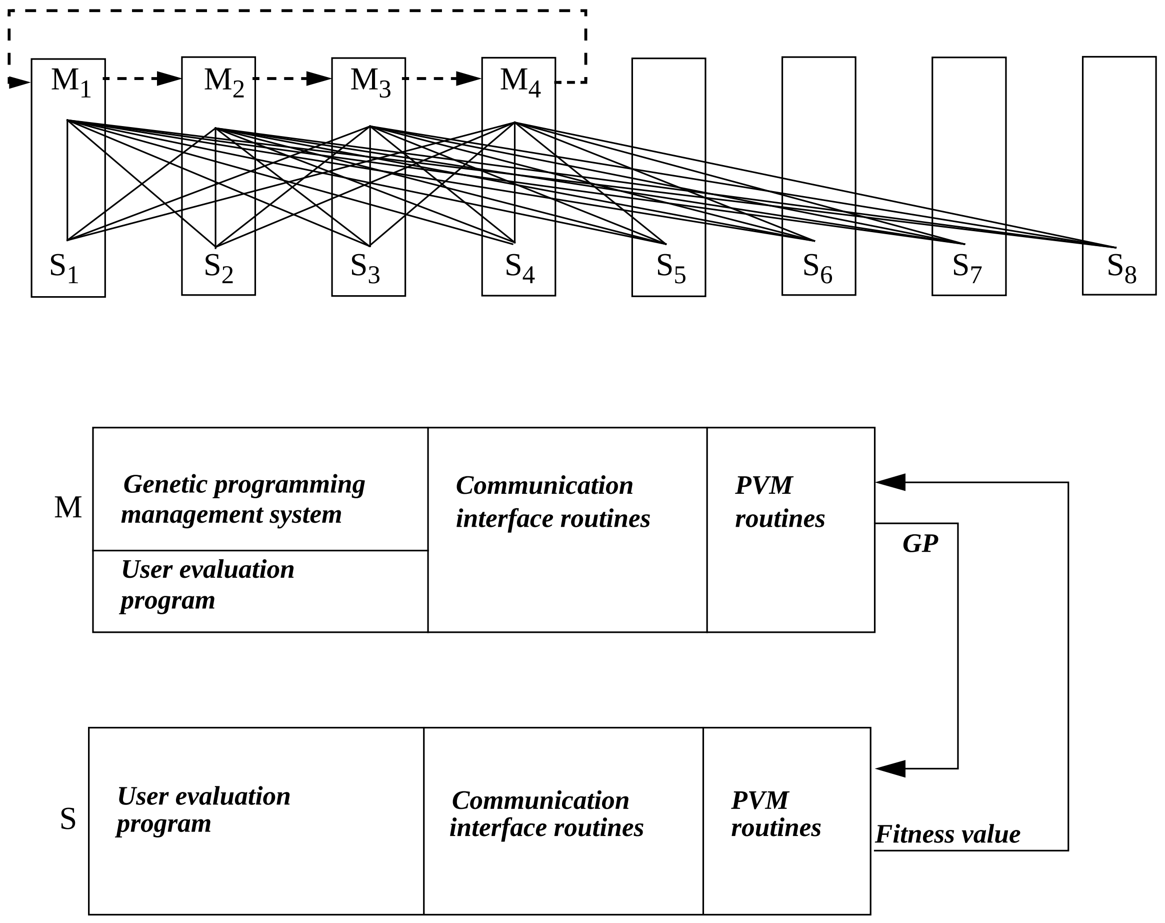


fig 15: Example of PGPS architecture evolving 4 subpopulations on 8 processors. There is a master (population management) process M_i for each subpopulation and each physical processor runs a slave process S_k which evaluates the individuals from any subpopulation. The solid lines show the communication pattern between master and slave processes. The dotted line shows the ring topology that allows individual migration across masters.

Each new genetic program is sent individually as soon as created. The messages belonging to the same destination are not clustered as a single message. This technique avoids delaying the fitness computations and allows the slave processes to be further kept busy. In this way, fitness computation and work distribution overlap.

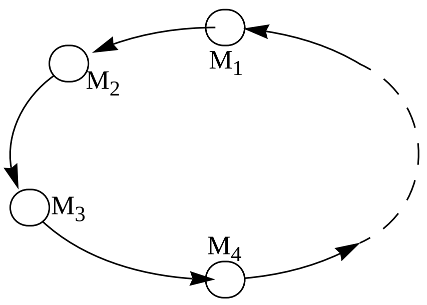


fig 16: Asynchronous task migration between master nodes using ring topology

The global evaluation ends when all subpopulations are evaluated (each processing node has performed its assigned tasks). Once evaluation completes, the subpopulations interact, to delay the convergence, using the ring topology. Each master node selects a genetic program and sends it to its next neighbor and receives asynchronously, using a nonblocking primitive, an individual from its previous neighbor (see figure 16). If the individual arrived then it is inserted in the subpopulation by replacing the least fit one, otherwise the node continues its evolutionary process. The termination signal of parallel execution is given when a pre-assigned maximum number of generations has been attained. The main steps of the parallel algorithm are shown in Figure 17.

Master process :

```
0: Load the other Master processes*.
1: Load the slave processes*.
2: Create the initial population.
3: Distribute the work load to the processing nodes.
4: Execute the local tasks.
5: Execute the external tasks (requested by other Master processes) and
   send back the results.
6: Receive the fitness values (sent either by a slave or a Master process).
7: Select and send an individual to the next neighbor
8: If an individual arrived from the previous neighbor then
   insert it in the subpopulation
9: Perform the selection phase.
10: Perform the reproduction phase.
11: Repeat Steps 3 - 8 until the maximum number of generations.
12: Terminate the slave processes*.
13: End.
```

(*) performed only by the first loaded Master process.

Slave process :

```
0: Receive a genetic program.
1: Calculate the fitness of the received program.
2: Send the fitness value to the source Master process.
3: Repeat Steps 0 - 2 until reception of termination signal.
4: End.
```

fig 17: A parallel algorithm for Genetic Programming.

The evaluation of all subpopulations (global evaluation) is performed by interleaving local and external tasks (threads). Multiple threads are maintained on each master node (steps 4 and 5) and switching among them overlaps message latencies by performing useful computation while other threads wait for synchronization signals. Multithreading a processing node is a method for improving processor utilization [1].

3.3 Data structures for the GP population

The system uses five main classes for representing in memory the whole population.

1. The class *Population*.
2. The class *GP*.
3. The class *Gene*.
4. The class *Function*.
5. The class *Terminal*.

The different relations between these classes are shown in figure 18.

- Relation R1: a *GP* is an individual of the *Population*.
- Relation R2: a *GP* is composed of *Genes*.
- Relation R3: a *Gene* may represent a *Function*.
- Relation R4: a *Gene* may represent a *Terminal*.

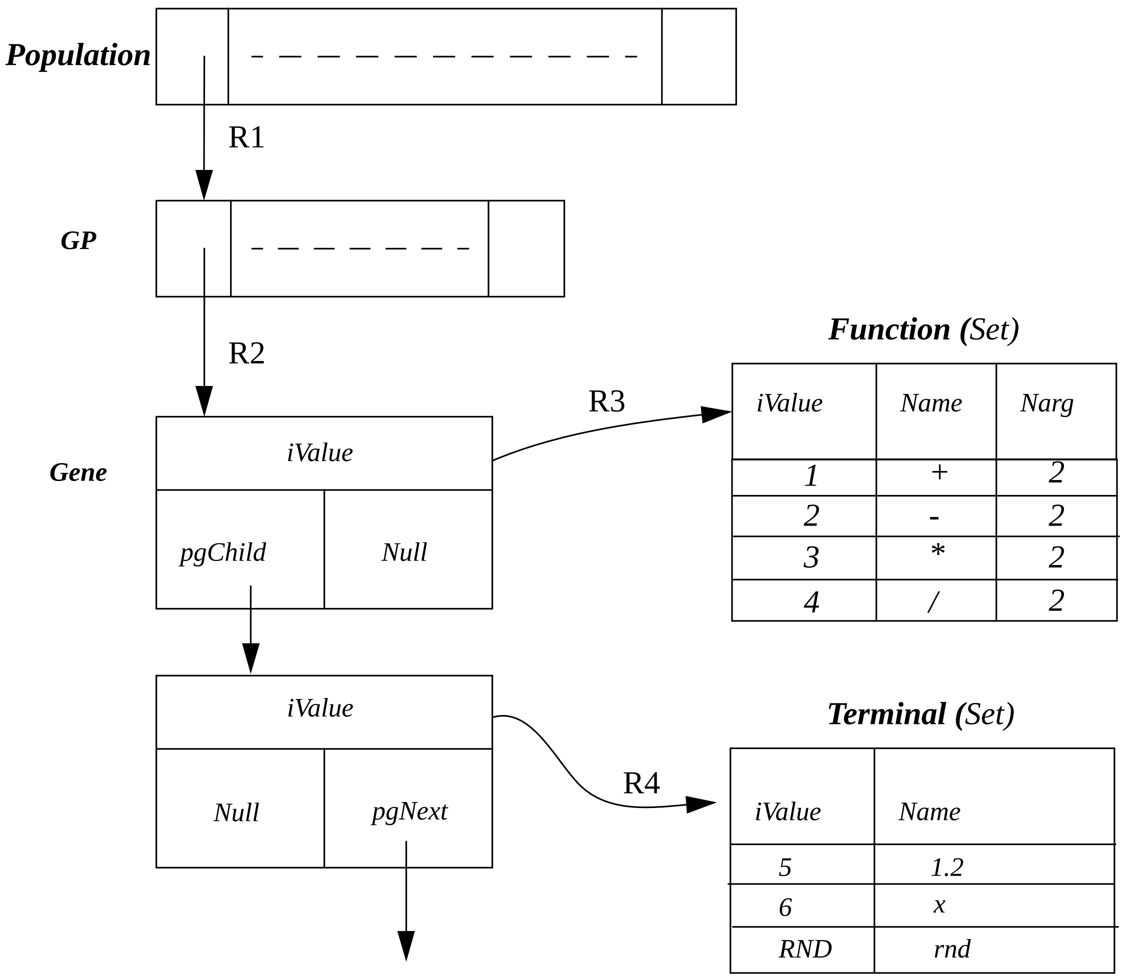


fig 18: Class hierarchy.

1- The population is defined by the class *Population* representing the individuals in a linear array. Each element of this array contains a pointer to an individual (genetic program).
Two attributes *uliFitness* and *uliLength* are used to store respectively the total fitness and total complexity of the population at each generation. The method *TotalFitness*() returns the total fitness of the population and updates the variable *uliFitness*. In similar way, the method *TotalLength*() returns the total complexity of the population and updates the variable *uliLength* .
The evaluation of the population is performed by the method *Evaluate*(). Furthermore, this function balances the workload on the processing nodes. The function member *Mutate*() performs selection and mutation of genetic programs. Similarly, the method *Generate*() performs selection and crossover. A random selection of a genetic program is achieved using the function member *Select*() which is called by the tournament selection.

2- The class *GP* defines a genetic program. A genetic program may be represented by several (usually one) parse trees (ADFs).

An ADF (Automatically Defined Function) refers to a function defined at run-time by a genetic program. Each defined function is associated to a single genetic program and is not visible by the whole population. Once defined, an ADF can be called by the genetic program as function or simply as terminal. The evolution of this dual structure (function definition and function calls) is driven by the fitness measure in conjunction with selection and genetic operations. Koza [6] asserts that the use of ADFs reduces the complexity of the solutions.

For the purpose of ADF handling, a linear array is provided to store the address of the root node of each parse tree. An element of this array contains a pointer to an object of class *Gene*. Note that if ADFs are not used then the GP will be represented by a single parse tree (the first element of the array).
Two data members *iFitness* and *iLength* are used to store respectively the fitness and complexity of the genetic program. The class GP provides function members that operate on an individual. Typically, these operations include packing / unpacking a genetic program , evaluating an individual over a set of fitness cases, mutating a GP or crossing it with another individual.

3- The class *Gene* defines a node of a parse tree. Each node of the parse tree represents a gene of the individual. A gene is characterized by an integer value, *iValue*, identifying a terminal or a function in the *TerminalSet* or *FunctionSet*. A function node requires at least one argument. The address of the first argument of a function is given by the pointer *pgChild*. The pointer *pgNext* is used to chain with the remaining arguments. Consequently, the attributes *pgChild* and *pgNext* are pointers to objects of class *Gene*. Note that this construction requires *pgChild* to be NULL for terminal nodes, as there are no arguments. The random numbers are stored in the attribute *forand*.

4- The class *Function* defines the *FunctionSet*. Each function introduced by the user will be an instance of this class. A function is identified by a numerical value *iValue*. The attribute *Name* is a character string containing the name of that function. Finally, the arity of the function is registered in the attribute *Narg*.

5- The class *Terminal* defines the set of terminals. Each terminal introduced by the user will be an instance of this class. A numerical identifier**,** *iValue*, is assigned to each terminal in the *TerminalSet*. The name of the terminal is stored in the attribute *Name*.
The declaration of classes *Population*, *GP* and *Gene* is given in *Annexe 1*.

3.4 Memory representation of a genetic program

Figure 19 shows the memory representation of the s-expression *((* (+ x x) (* x x)))*.

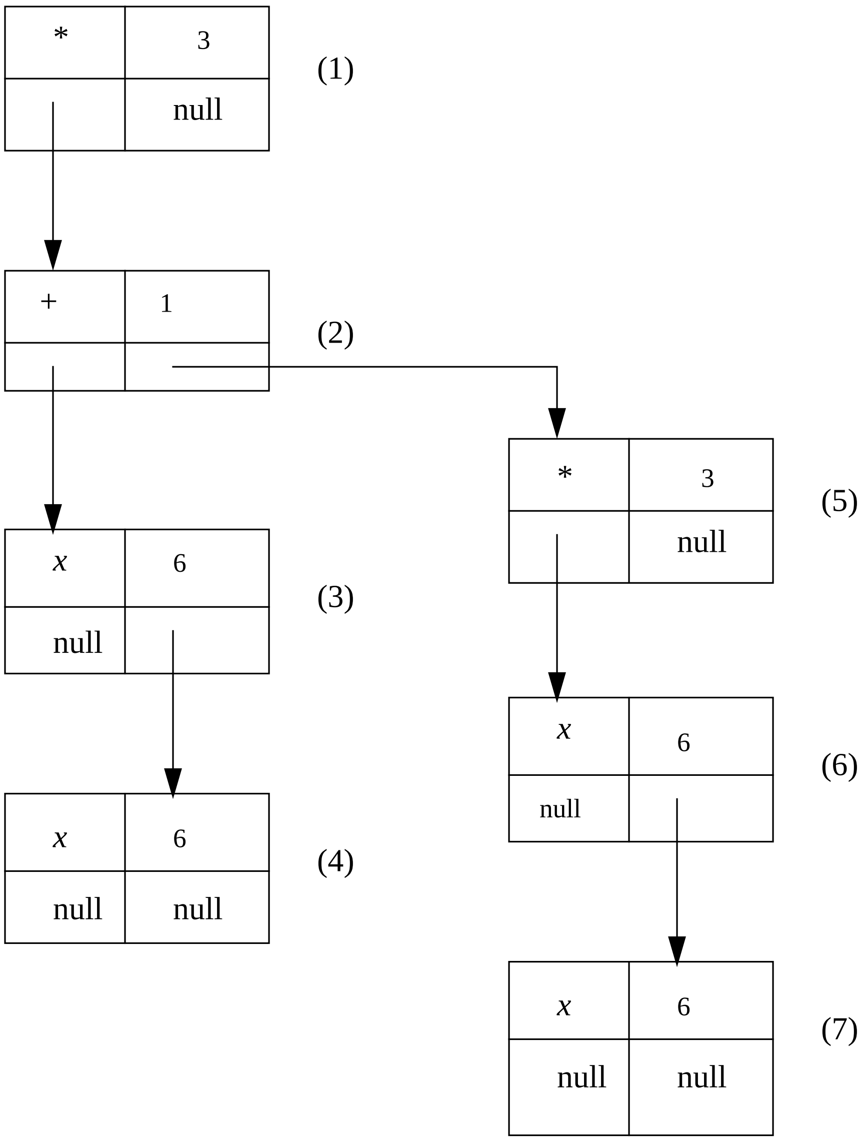
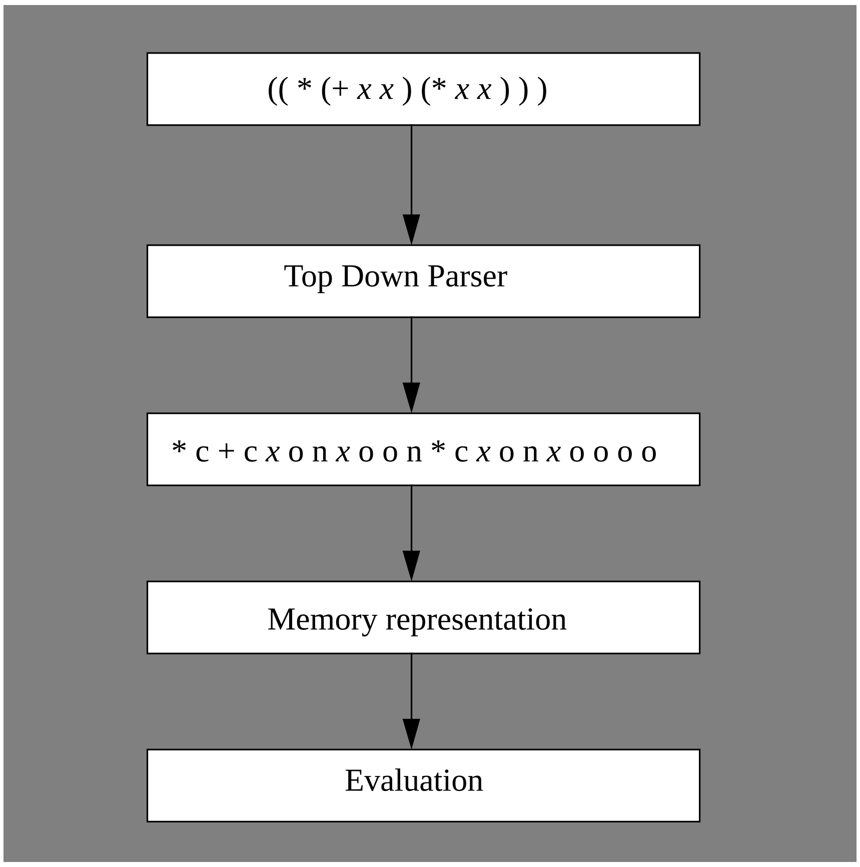


*fig 19: Memory representation of ((* (+ x x) (* x x)))*

Node 1 (the root node) is a function node representing the operator '*'. The address of the first argument (node 2) of this function is given by the left pointer (or *pgChild*) and node 5 represents its the second argument. The right pointer (or *pgNext*) of node 2 gives the address of node 5. Node 2 happens to be a function node representing the operator ‘+‘. The left pointer of this node contains the address of the first argument (node 3) which is chained with the second argument (node 4) via the pointer *pgNext*. In similar way, node 5 is the root node of the subexpression *(* x x)* . The pointer *pgChild* of this node is equal to the address of the first argument (node 6). Finally, the pointer *pgNext* of node 6 contains the address of the second argument (node 7). We remark that the arguments belonging to the same function are grouped in linear list which makes flexible the representation of multiple functions with different number of arguments in the same genetic program.

3.5 Evaluating genetic programs

Before fitness calculation, a genetic program is first transformed from the memory representation into a linear form (character string) and then transmitted as a message to a slave process. When received, the genetic program requires to be reconstructed before the evaluation. Figure 20 depicts the transformations of a message from the Lisp syntax to its memory representation. The S-expression is first parsed using a top down parser in order to generate an intermediate form suitable to memory representation. In theory of languages, an intermediate form is a language somewhere between the source high-level language and machine language [2]. The extra-symbols ‘c‘ , ‘n‘ and ‘o‘ indicate respectively the first argument (*pgChild*) , next argument (*pgNext*) and the *NULL* pointer value. This intermediate form is then exploited to build the memory representation of the genetic program. The obtained parse tree is evaluated as many times as there are fitness cases related to the problem at hand.



((* (+ x x) (* x x)))
Top Down Parser
* c + c x o n x o o n * c x o n x o o o o
Memory representation
Evaluation

fig 20: Translation of genetic programs.

3.6 Parsing genetic programs

In order to perform the syntax analysis of a genetic program and also to generate the corresponding intermediate form, the system uses the Syntax-Directed Translation based on a recursive descent parser. Figure 21 shows the top down parser presented as finite state machine.

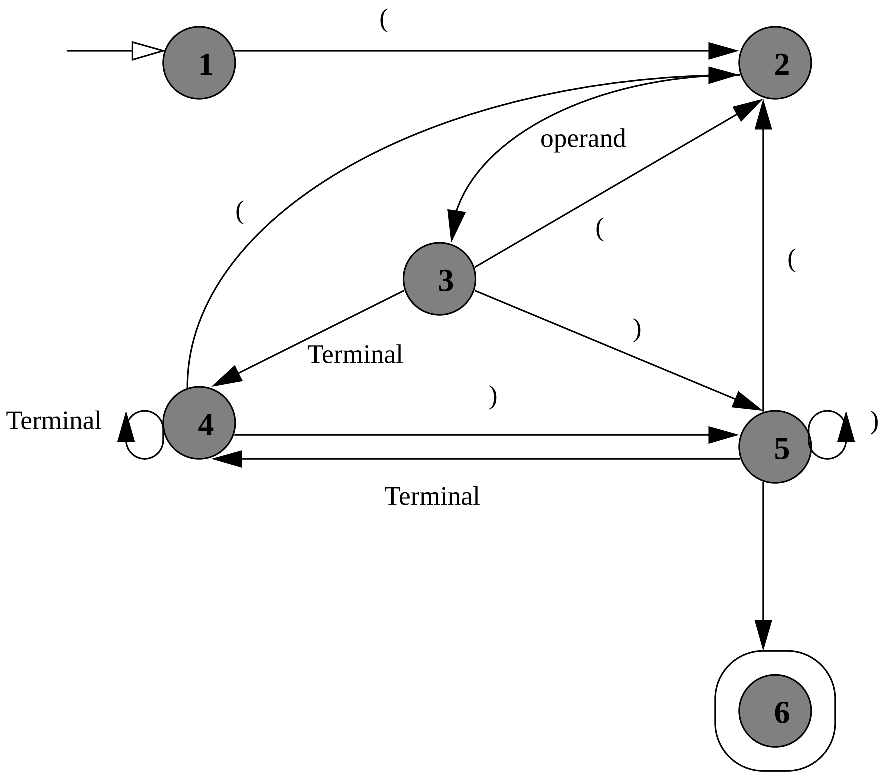


fig 21: Top down parser for genetic programs.

Each state is labeled by a number and each transition is labeled by a lexical token. The initial state and the final state are indicated respectively by the numbers 1 and 6. While parsing the input genetic program, each transition calls an underlying semantic action. Figure 22 summarizes the different transitions followed by their semantic actions enclosed in curly braces {}. The variable i counts the number of parenthesis currently opened and the variable Operand may be a terminal or function name.

Transition I : J	Semantic action
1 : 2	{ i=1 }
2 : 3	{ Print(Operand) }
3 : 2	{ i++; Print(‘c’) }
3 : 5	{ i--; Print(‘o’) }
3 : 4	{ Print(‘c’); Print(Terminal); Print(‘o’) }
4 : 4	{ Print(‘n’); Print(Terminal); Print(‘o’) }
4 : 2	{ i++; Print(‘n’) }
4 : 5	{ i--; Print(‘o’) }
5 : 5	{ i--; Print(‘o’) }
5 : 4	{ Print(‘n’); Print(Terminal); Print(‘o’) }
5 : 2	{ i++; Print(‘n’) }
5 : 6	{ if (i == 0) return }

fig 22: Transition semantic actions.

3.7 Creation of the initial population

Each run of PGPS starts with the creation of a population of random computer programs, each composed from the available functions constituting the function set and the available terminals constituting the terminal set. The system uses two parameters for controlling the creation process of a random genetic program: the Maximum Depth For Creation (MDFC), which is the maximum depth that a GP may reach at the creation phase, and the creation method. Five different methods are used by the system for creating the initial population. These methods are the variable method, the grow method and three ramped methods. Each of these methods chooses a function, rather than a terminal, to be the root node of each GP at creation phase.

3.7.1 Variable method

This method consists of creating genetic programs whose the depth varies from 1 to a maximum value (MDFC).

Example:

> *MDFC = 2;*
> *FunctionSet = { + };*
> *TerminalSet = { T }.*

The possible genetic programs that will be generated using the variable method are:

((+ T T))	*, depth = 1;*
((+ (+ T T) T))	*, depth = 2;*
((+ T (+ T T)))	*, depth = 2;*
((+ (+ T T) (+ T T)))	*, depth = 2.*

3.7.2 Grow method

This method consists of creating genetic programs whose the depth is MDFC. All the individuals will reach the maximum depth and also the maximum structural complexity (if all functions, in the function set, have the same arity).

Example:

MDFC = 2;
FunctionSet = { + , - };
TerminalSet = { T }.

The possible genetic programs that will be generated using the grow method are:

(([+ | -] ([+ | -] T T) ([+ | -] T T))) , depth = 2.

where [+ | -] indicates one of the ‘+’ or ‘-’ function.

3.7.3 Ramped methods

The population is divided into groups. To each group is assigned a local maximum depth (allowable depth). The value of the allowable depth varies progressively from 2 (the allowable depth for the first group) to the maximum allowable depth (for the last group).

The size of a group (*GroupSize*) is given by:
GroupSize = 1 + (PopulationSize / (MDFC - 1)).
Where ‘/’ indicates the integer division.

Example:

PopulationSize = 10;
MDFC = 6.

We get the size of each group by applying the calculation above:
GroupSize = 1 + 10 / (6 - 1) = 3.

The population will be partioned as follows:

group0 = {0, 1, 2}	*, allowable depth = 2;*
group1 = {3, 4, 5}	*, allowable depth = 3;*
group2 = {6, 7, 8}	*, allowable depth = 4;*
group3 = {9}	*, allowable depth = 5.*

The system uses 3 ramped methods based on the population partitioning described above.

- Ramped variable method

This method applies the variable method to each group. A group is viewed as subpopulation which has a size (GroupSize) and a maximum depth (allowable depth).

- Ramped grow method

This method applies the grow method to each group. Inside a group, all genetic programs will reach the allowable depth assigned to that group.

- Ramped half-and-half method

The ramped half-and-half method is the most frequently used in genetic programming. This method combines the variable and the grow methods. Inside each group, half of the individuals are created using the variable method and the other half using the grow method.

3.8 Interaction with the user

Runs of PGPS are controlled by parameters read from an initialization file. Figure 23 illustrates the configuration of the initialization file. The list of the control parameters is hereafter presented and the meaning of each parameter is highlighted. The names used to identify each parameter are self explanatory.

Function Set : Indicates the list of functions constituting the function set. The functions are separated by comma and the list is ended by a semicolon. Each function, in the function set, is described by a name followed by its number of arguments (arity) enclosed in parenthesis.

Terminal Set : Indicates the list of terminals composing the terminal set. The terminals are separated by comma and the list is ended by a semicolon. Each terminal, in the terminal set, is described by a name. The random terminal is described as rnd(min,max) where (min,max) indicates the range of the random values that will be generated at the population creation phase or at the reproduction phase (by mutation).

PopulationSize : Indicates the size of the population.

Generations : Indicates the number of generations that will be performed by the system during a run. The system returns after the last generation. This parameter is also used as the stop condition by the master process.

Creation Type : This parameter identifies the creation method that will be used by the system during a run. The possible values are between 0 and 4.

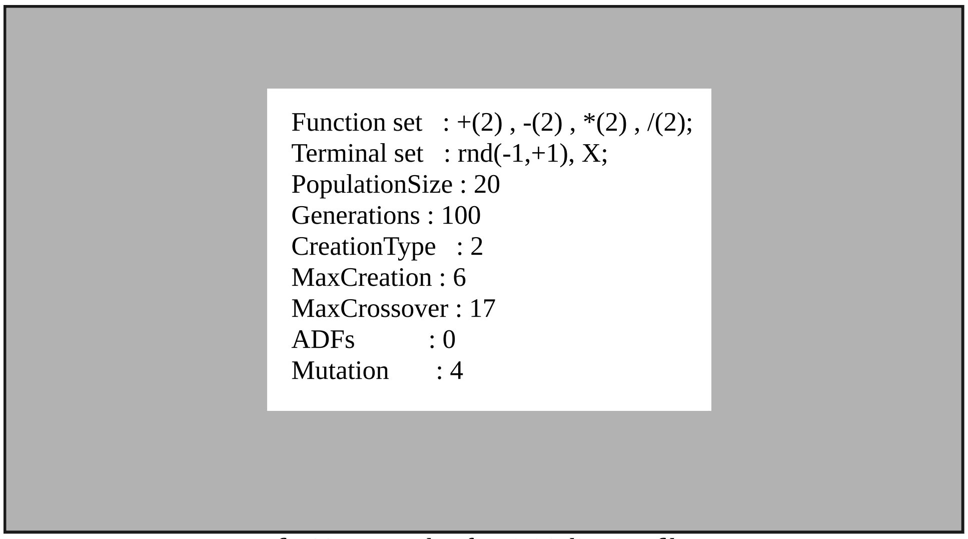
0 : selection of the Variable method;
1 : selection of the Grow method;
2 : selection of the Ramped Half and Half method;
3 : selection of the Ramped Variable method;
4 : selection of the Ramped Grow method.

MaxCreation : This value is the maximum depth for creation (MDFC). It is the maximum depth that a genetic program can reach at the population creation phase.

MaxCrossover : This value is the maximum depth for the crossover operation. It is the maximum depth that a genetic program can reach at the reproduction phase.

ADFs : Indicates the number of ADFs that will be used to solve the problem. A value of 0 is used for runs without Automatically Defined Functions (ADFs).

Mutation : This value controls the mutation rate. The probability of mutation operation is given by the (*Mutation / PopulationSize*) ratio.



```
Function set   : +(2) , -(2) , *(2) , /(2);
Terminal set   : rnd(-1,+1), X;
PopulationSize : 20
Generations : 100
CreationType   : 2
MaxCreation : 6
MaxCrossover : 17
ADFs          : 0
Mutation      : 4
```

fig 23: Example of an initialization file.

The execution output is saved in data files containing the best individual at the end of each generation and also some statistics relying with the population fitness and complexity which can be interactively visualized using the graphical tool shown in *Annexe* 6.

3.9 Application of PGPS to the logistic function

We have studied the ability of PGPS to forecast time series, which is an important problem in economics, meteorology, and many other areas.

The logistic map is one of the simplest models of a system exhibiting deterministic chaos. The term chaos is used to describe a complex behavior of some systems [5].

Time series generated by the logistic map are given by :

$$x(t+1) = F[x(t)] \quad \textit{with} \quad F[x] = 4x(1-x) \quad , \quad 0 \le x \le 1 \quad .$$

The function was defined as a set of $n = 500$ points : $\left(x_i \ , F[x_i]\right)$, $i = 0, ..., n-1$

The parameter file used for solving this problem is identical to the one illustrated as example in figure 23. The *TerminalSet* will include the variable *x* and, in what follows, we show the results of runs with and without including random numbers in the *TerminalSet*. Each genetic program will be expressed as a function, *GP[x]*, of the terminal variable *x*.

3.10 The fitness function

The fitness measure of a GP, that we have used to solve this problem, is the minimization of sum square deviation between correct and forecast values:

$$min \quad \left(\sum_{0}^{n-1} (GP(x_i) - F(x_i))^2\right), \ (n = 500)$$

In terms of maximization of the fitness function, this is equivalent to

$$max \quad -\left(\sum_{0}^{n-1} (GP(x_i) - F(x_i))^2\right), \ (n = 500)$$

Note that the logistic function has two fixed points: F(0) = 0 and F(3/4) = 3/4.
In this application, we have chosen $x_0 = 0.2$ as initial value.

3.11 Sufficiency of the *FunctionSet* and *TerminalSet*

Three sets of runs are done. The first series of runs uses the random integer constants (N^*) , the second uses the floating point random constants between -1.00 and +1.00 and the third does not include random constants.

- Using the integer type for the ephemeral random constant in the terminal set, we present here, the results of the best run. By generation 57, the fitness of the best-of-generation program was -9.2071. This individual has 27 function and terminal nodes and is shown below.

((- x (x (* (* x (* x (/ x (* x x)))) (* x (+ x (- x (/ x (* x (* x x))))))))))) .*

This genetic program, when simplified, is equivalent to :

$$GP[x] = -2x^4 + 2x \quad .$$

Figure 24 compares the genetically evolved best-of-run program, *GP[x]*, and the correct function. As can be seen, this individual bears some resemblance to the logistic map function *F[x]*.

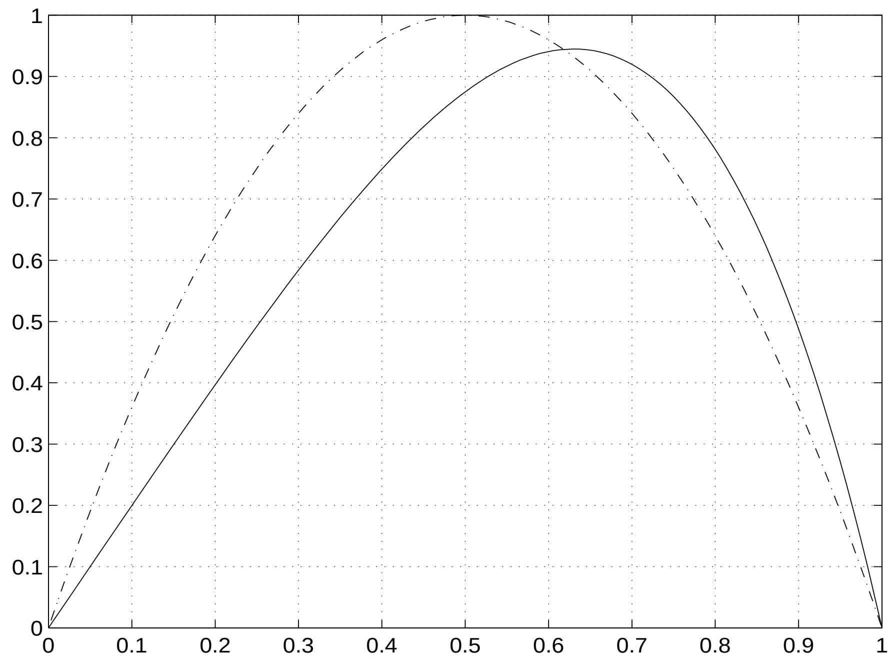


fig 24: Best of run program GP[x] (solid line) using random integer constants compared with the logistic map function F[x] (dashed line).

- Changing the type of the random numbers to be of the floating point type between -1.00 and +1.00 seems to improve the solution. In Generation 90, the sum of the squared errors for the new best-of-run individual improved to -3.79298. This is approximately a 2-to-1 improvement of the fitness. This individual has a structural complexity of 29 :

*((/ x (+ (* x (* x (* x (+ x (* x (* x (* x (* x (+ x (+ x (* x (/ x 0.37)))))))))))) 0.38)))*

This S-expression is equivalent to

$$GP\,[x] \;=\; \frac{x}{2 \cdot 7027x^{9} + 2x^{8} + x^{4} + 0 \cdot 38}$$

Figure 25 compares *GP[x]* with *F[x].*

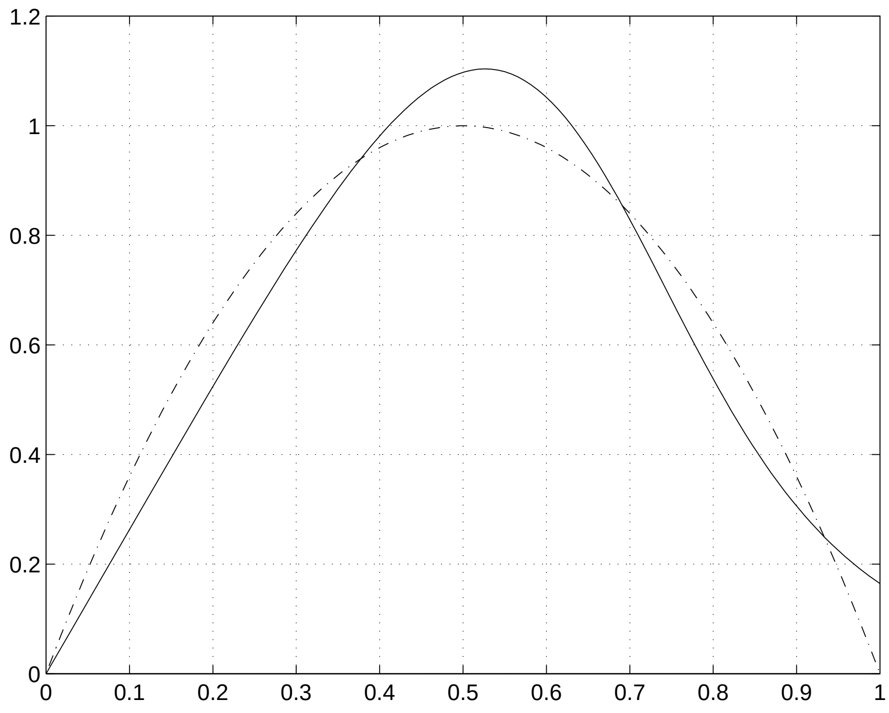


fig 25: Best-of-run S-expression using the ephemeral random floating-point constant ranges over the interval [-1.00 , +1.00]. The dashed line curve is the logistic map function F[x] and the solid line represents the best -of-run program GP[x].

Figure 26 shows a graph of the values of the time series produced by the best-of-run individual from generation 90 overlaid onto a graph of the values of the logistic map function. The graph runs from time step 500 to 550 so as to show the behaviour of the function *GP* beyond the range defined by the 500 fitness cases. The initial value for both time series was

x_{n-1} .The two sequences were somewhat close on time steps 501 and 502. Thereafter, the sequence generated by *GP* diverged considerably from the correct values of the logistic sequence.

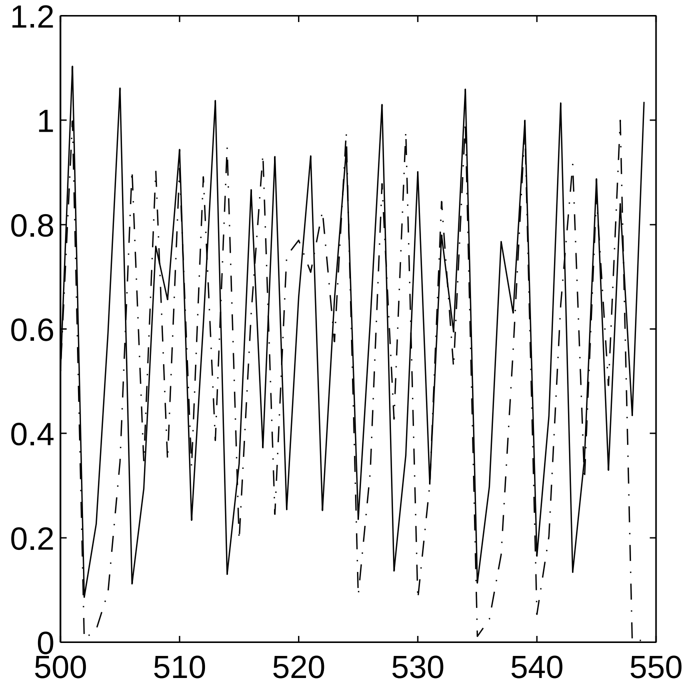
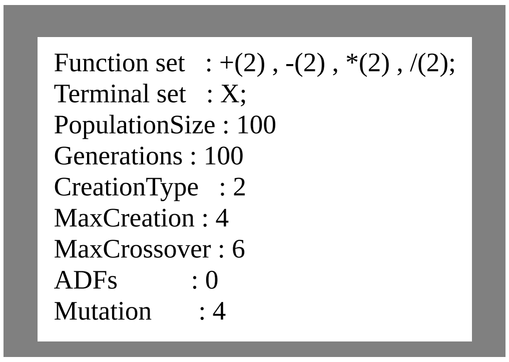


fig 26: Out-sample behaviour of the time series for the correct logistic function (dashed line) and the best-of-run program GP[x] (solid line) .

- Without using random constants and taking the parameter file presented in figure 27 leads to a successful run.



```
Function set   : +(2) , -(2) , *(2) , /(2);
Terminal set   : X;
PopulationSize : 100
Generations : 100
CreationType   : 2
MaxCreation : 4
MaxCrossover : 6
ADFs          : 0
Mutation       : 4
```

fig 27: Parameter values of a successful run solving the logistic function problem.

The best-of-generation individual emerging in generation 28,

((/ (- (+ x (- x (* x x))) (* x x)) (/ x (+ x (- x (- x x)))))) ,

had a perfect fitness value of **-1.01873e-29** . This individual, when simplified corresponds precisely to the logistic function : $4x(1-x)$.

From generation 54 to 100, all the best-of-generation programs scored the fitness value of -1.01873e-29 and also contain the building block given by ***(- (+ x (- x (* x x))) (* x x))*** . The most parsimonious best-of-generation program was found at generation 76 and is of complexity 17 :

((/ (- (+ x (- x (* x x))) (* x x)) (/ x (+ x x))))

Figure 28 shows, by generation, the population average complexity and the fitness of the best-of-generation individual.

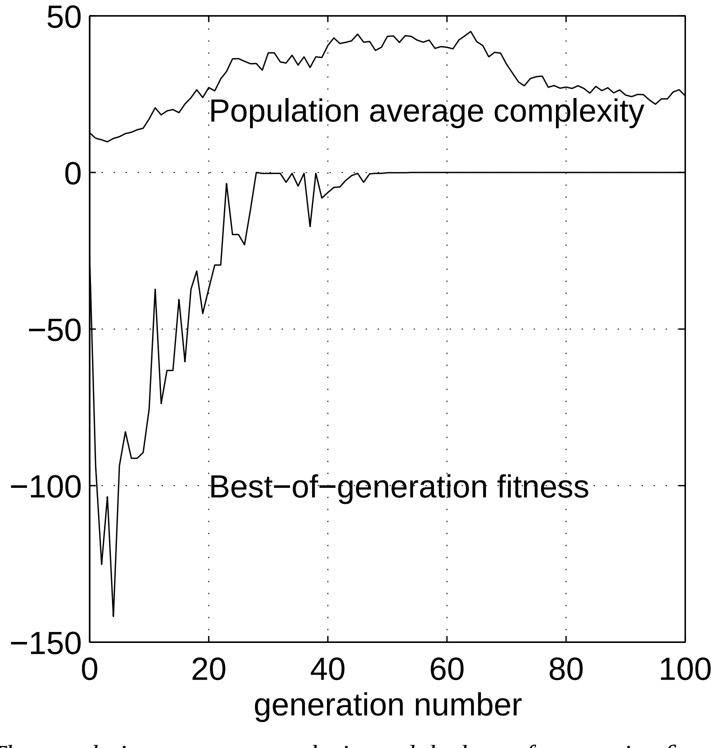


fig 28: The population average complexity and the best-of-generation fitness curves.

Bibliography

[1] Agarwal, A. 1992. Performance Tradeoffs in Multithreaded Processors, IEEE Transactions on parallel and distributed systems, vol. 3, no. 5, September 1992.

[2] Bergmann, S. 1994. Compiler Design: Theory, Tools and Examples, WCB Publishers.

[3] Fraser, A. P. 1994. Genetic programming in C++, public domain genetic programming system.

[4] Geist, A., Beguelin A., Dongarra J., Jiang W., Manchek R. and Sunderam V. 1994. PVM3 User's Guide and reference manual, ORNL/TM-12187, September 1994.

[5] Hilborn, R. C., 1994. Chaos and Nonlinear dynamics, Oxford University Press.

[6] Koza, J. 1994. Genetic programming II, Automatic Discovery of Reusable Programs, MIT Press.

[7] Oussaidene, M., Chopard B., Pictet O. V. and Tomassini M. 1996. Parallel genetic programming: an application to trading models evolution. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996*, pages 357-362. MIT Press, 1996. Proceedings of the first Annual Conference, July 28-31, 1996, Stanford University.

[8] Oussaidène, M., Chopard B. and Tomassini M. 1995. Programmation évolutionniste parallèle, RenPar'7 , PIP-FPMs Mons , Belgium. Libert G., Dekeyser J.L., Manneback P. (editors).

[9] Oussaidene, M., Chopard B., Pictet O. V. and Tomassini M. 1996. Parallel Genetic Programming and its application to trading model induction. Submitted to the journal *Parallel Computing*, July 1996 .

4. Complexity and Scalability Analysis

Section 4.1 analyzes the time complexity of a PGPS sequential run. In section 4.2 we propose two different schemes for load balancing. The standard metrics (speedup and efficiency) for evaluating the performances of a parallel algorithm are recapitulated in section 4.3. The system scalability and the execution time model are analyzed in section 4.4. The performance evaluation of PGPS is described in section 4.5. Finally, the characteristics of the IBM SP-2 machine are shown in section 4.6.

4.1 Complexity analysis

When parallelizing a serial application, it is important to identify the parts of the application that take the most time to run. These are known as hot spots. Since improving performance involves reducing the processing time of the most compute-intensive parts of the application, parallelizing them is a good way to achieve this improvement. This technique is known as hot-spot analysis.
Let be :

p : the population size;
g : the number of generations;
n : the number of fitness cases;
d : the maximum depth for crossover (MDFC);
r : the maximum function arity in the function set .

In this analysis, the problem size is given by the number of fitness cases, n, which is problem dependent. All the other parameters p, g, d indicating respectively the population size, number of generations and maximum depth are genetic programming parameters.

In order to identify the parts of the system that take the most time to run, figure 29 shows a profile of a sequential run of PGPS where each GP is evaluated 1000 times. In this execution, we used the values $p = 50$, $g = 100$, $n = 1000$ and a maximum depth of $(d = 6)$. The total elapsed time of this run was 296.27 seconds on one SP-2 node and the total elapsed time

required for evaluating the population was 290.43 seconds. Therefore, 98 % of the sequential execution time of the system is used during the evaluation phase.
We can remark fluctuations in the population evaluation curve. These fluctuations are due to fact that genetic programs do not have a static structure and their complexity varies dynamically at run time.

Note that all the times reported were measured in the dedicated mode (the code runs in a single user mode on the processing node).

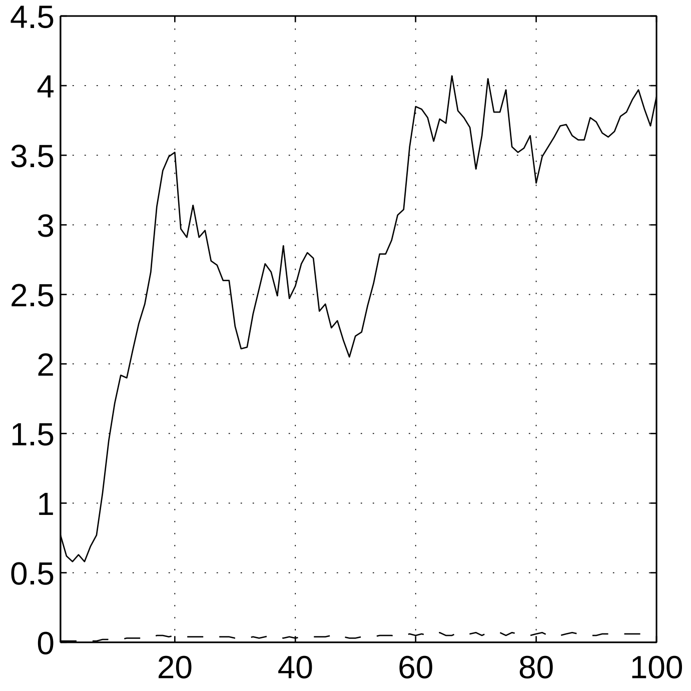


fig 29: CPU time (in seconds) required at each generation for the reproduction phase (dashed line) and the evaluation phase (solid line).

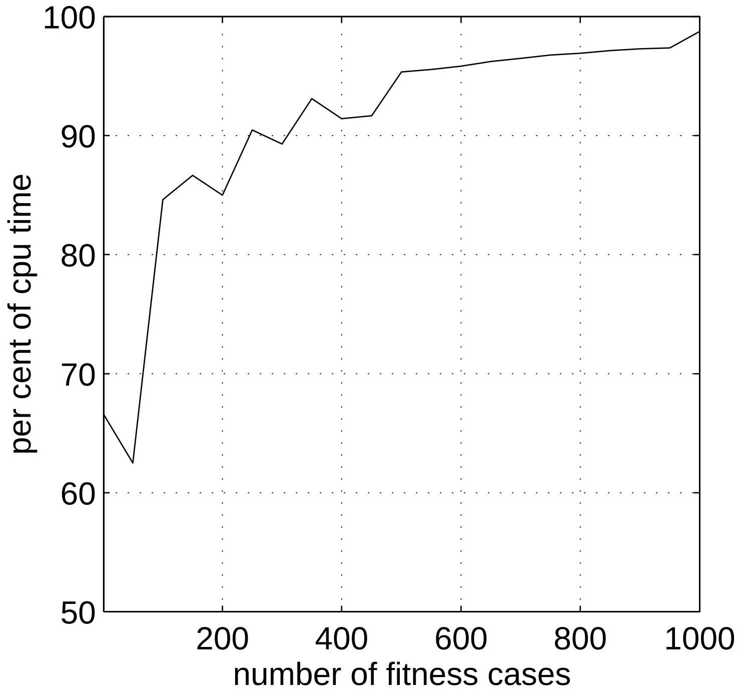


fig 30: The population evaluation time in relation with the number of fitness cases, $(p, g, d) = (10, 10, 6)$

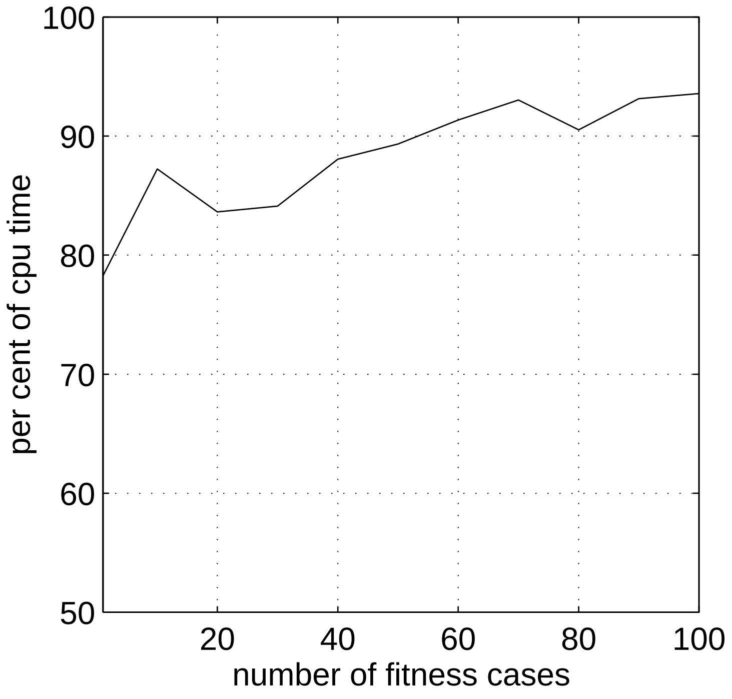


fig 31: The population evaluation time in relation with the number of fitness cases, $(p, g, d) = (10, 10, 12)$

Figure 30 and figure 31 show the time spent during the evaluation phase, expressed in per cent of the sequential execution time, when increasing the number of fitness cases n. Both population size and number of generations parameters are set to 10. We remark that for less complex genetic programs $(d = 6)$, the evaluation phase consumes more than 90% of the sequential execution time when the number of fitness cases, n , exceeds 400. However, for more complex GPs $(d = 12)$, the population evaluation time reaches 90% of the sequential execution time around $n \geq 60$. This is due to fact that the population evaluation time grows in exponential way ($\theta(r^{d+1})$) with the maximum depth (according to formula 4.1) and in linear way ($\theta(n)$) with the number of fitness cases .

For problems of large enough size, the evaluation phase takes the most time to run and thus represents a potential source of parallelism and then making the sequential part (i.e the population management) of the application practically negligible. The execution time can be assimilated to the evaluation time. We highlight, hereafter, the main parameters that control the elapsed time of a sequential execution.

The crucial factors that affect the time of a sequential run of PGPS system are the population size, complexity of the population, number of generations and the number of fitness cases . We discuss, hereafter, the impact of these parameters upon the whole execution time.

- *Population Size* : The population size parameter , *p* , is the number of individuals that the system handles at each generation. Let t_i be the evaluation time of an individual *i* . If all the individuals in the population were identical to the individual *i* , then evaluating the whole population would require a time of $p \cdot t_i$. This time is also equivalent to evaluating *p* times the same individual *i*. Thus, the evaluation time of the population is proportional to the Population Size parameter.

- *Population Complexity* : The complexity of a genetic program is defined as the number of function and terminal nodes in the associated parse tree. The more an individual is complex, the more it has operations to execute and , thus , more it takes time for evaluation . The complexity, C_j , of an individual , *j* , can be bounded by the following formula.

$$\left\{ \begin{array}{ll} 1 \leq C \leq \dfrac{r^{d+1} - 1}{r - 1} & \text{if} \quad (r \neq 1) \\ & \\ 1 \leq C \leq d + 1 & \text{if} \quad (r = 1) \end{array} \right\} \qquad \textbf{(4.1)}$$

Where *d* is the maximum depth for crossover and *r* is the maximum function arity in the function set .

Each individual , *j* , in the population needs C_j operations to be evaluated. Evaluating a population of *p* individuals requires $(C_1 + \ldots + C_p)$ operations . If $C*$ is the average complexity of the population , then the system will perform $pC*$ operations for the population evaluation.

- *Number Of Generations* : The number of generations , *g* , is the number of times that the system performs the selection , reproduction and evaluation phases. If C_i* is the average complexity of the population at generation *i* , then evaluating a population , of size *p* , during *g* generations will take $p(C_0* + C_1* + \ldots + C_g*)$ operations. C_0* is the average complexity of the initial population .

- *Fitness cases* : Fitness cases are related to the problem to solve. They are, in general, a small finite sample of the entire domain space and must be representative of this domain as a whole, because they form the basis for generalizing the results obtained to the entire search space. Each individual is evaluated n times during the fitness calculation . Since this parameter is fixed for the whole population (at the system initialization) and does not change during the run, it will scale the number of operations performed by the system when applied to a problem having only one fitness case $(n = 1)$. Note that the number of fitness cases is also a metric for the problem size.

Finally , a sequential run of PGPS , solving a problem of n fitness cases, evolving a population of size *p* , over a number of generations *g* , will require a time T_{seq} which can be approximated by :

$$T_{seq} \cong n \cdot p \cdot \left(\sum_{i=0}^{g} C_i^* \right) \cdot t_a \qquad \textbf{(4.2)}$$

where t_a is the cost of one arithmetic operation.

From experimental performance measurements on the IBM-SP2 machine, we derive

$$t_a = 1,44 \times 10^{-6} \text{ seconds}$$

for a mixture of standard arithmetic operations.

Therefore ,

$$T_{seq} \cong n \cdot p \cdot \left(\sum_{i=0}^{g} C_i^* \right) \cdot (1,44 \times 10^{-6}) \qquad \sec onds .$$

Figure 32 shows the logarithmic representation , for each generation $i \in [0,100]$, of the population average complexity curve overlaid onto the required cpu-time for evaluating the population at generation *i* . The generation 0 is the population creation phase . The different parameter values for this run were *p = 50 , g = 100 , n = 1000 , d = 6* and *r = 2* .

Let T_i be the population evaluation time at generation *i* .
From the previous formula, we get :

$$T_i \cong 1000 \cdot 50 \cdot \left(1,44 \times 10^{-6} \right) \cdot C_i^* \qquad \sec onds \ .$$

or

$$T_i \cong 0,072 \cdot C_i^* \qquad \sec onds \qquad i \in [0,100] \ .$$

Thus , we can remark that the population evaluation time curve is proportional to the population average complexity curve with a scaling factor of 0.072 .

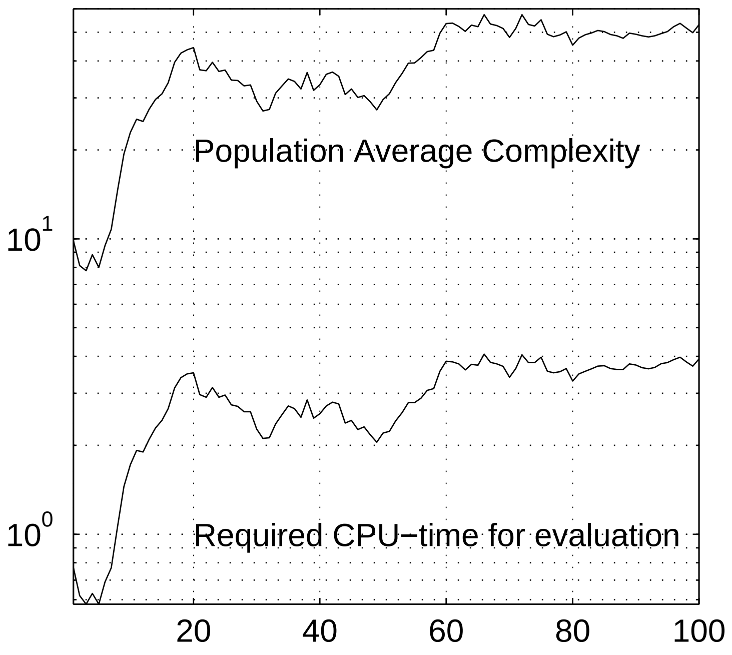


fig 32: Evolution, by generation, of the CPU-time (in seconds) required for evaluating the population and the population average complexity. The X-axis indicates the generation number.

4.2 Load Balancing

This section discusses the load balancing problem. Load balancing refers to equal distribution of the computational load among the processing nodes. It is an important performance enhancer when writing a parallel application. A parallel execution can be viewed as a collection of many cooperating tasks and, by optimal task distribution upon the processors, the parallel execution can deliver significant speedup. In the terminology used here, a task is defined as an atomic step sequence which :

- parses a genetic program ;
- builds the memory representation ;
- evaluates the GP on a set of n fitness cases , according to a specified fitness function ;
- returns the fitness value .

The task size refers to the complexity of GP to be evaluated. Note that this definition of task size refers also to the number of arithmetic operations required for a single evaluation of the GP. A task is the smallest unit that can be scheduled on a processing node. At the evaluation phase, the work load seen by any processor consists, essentially, of a set of tasks. Each task can be scheduled independently and is capable of being executed by any processor. Furthermore, once initiated, each task runs uninterrupted until termination. After task termination, each processor activates a task taken from its local wait queue according to FIFO policy. Load balancing can be either static or dynamic.

4.2.1 A static scheduling algorithm

This algorithm distributes the work load in an ordered way defined at compilation time. The Round-Robin policy is used to assign tasks to the processing nodes. The criterion of load balancing upon the processors is the number of tasks assigned to each processor. Task i , in the task pool to be distributed over m processors, is assigned to the processing node given by $\lceil i/m \rceil$.

This algorithm handles only static parameters, fixed at compilation time, and does not reflect dynamic factors which may change during the system evolution at run time. Even though, the algorithm regulates the number of tasks assigned to each processor, irregularity in computational load may occur. This overhead is due to fact that, tasks are not of the same size and, the work load on a processing node depends not only on the number of tasks to be performed, but also on the size of these tasks. This irregularity at complexity level induces differences in the task evaluation time which may unbalance the work load on the processing nodes. Figure 33 illustrates the mechanism of the static scheduling algorithm.

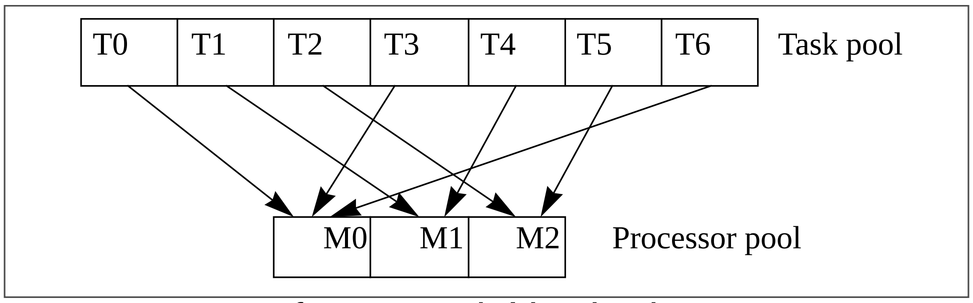


fig 33: A static scheduling algorithm

Static scheduling is appropriate for fixed work load environments. If the work load is known beforehand and does not change, optimal static scheduling algorithms can be found and implemented. Note also that static scheduling schemes are simple to implement and incur little extra overhead.

4.2.2 A dynamic scheduling algorithm

With this dynamic load balancing scheme, the decision of which processor should run a specified task is made at run time. The processor allocation is a function of population complexity at each generation of the evolutionary process. The work load on a processing node is considered as the sum of each task size assigned to that node. It is the single parameter which varies at run time and all the other parameters (p , g , n) are fixed for each run. Increasing the complexity of a GP enlarges the task size and therefore expands the task evaluation time. The basis of this dynamic scheduling algorithm is to balance, at each generation, the distribution of arithmetic operations upon the processing nodes and thereby preventing a situation where one or more processors falls idle. This problem can be formulated as follows.

Given m processing nodes $M_i \quad (i = 1, \ldots, m)$ and p tasks $T_j \quad (j = 1, \ldots, p)$, each with a size C_j , one wishes to find a schedule of minimum length. A schedule is optimal if the maximum task completion time is minimum. Note that this problem is known to be NP-hard. Let l_i be the work load of processor *i* . The algorithm is shown in figure 34. It is based on a “greedy” heuristic giving, in average, a good approximated solution to the problem.

```
Step 0: Sort $C_j$ in Downward order
Step 1: $l_i \leftarrow 0 \qquad (i \in [1 \ldots m])$
Step 2: For $j \leftarrow 1$ to $p$ do
            $i^* \leftarrow min\{i | l_i \le l_k, 1 \le k \le m\}$
            Assign $T_j$ to $M_{i^*}$
            $l_{i^*} \leftarrow l_{i^*} + C_j$
Step 3: End
```

fig 34: A dynamic scheduling algorithm.

The tasks are sorted by size in downward order. The task distribution starts from the largest task. At each iteration, the algorithm assigns the current task to the least loaded processing node. When a processor is selected to perform a task, then its work load is increased by the size of that task. Figure 35 shows an example of distributing 7 tasks on 3 processors.

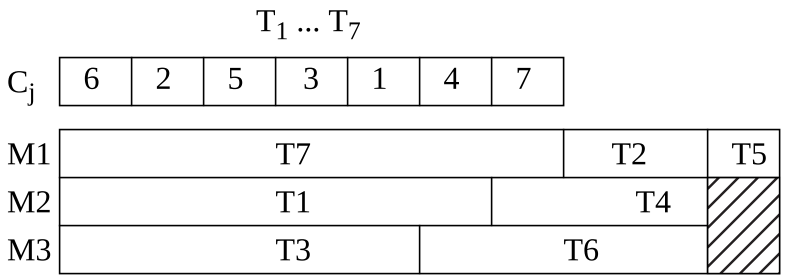


fig 35: An example illustrating the dynamic load balancing algorithm

The data structures required in the implementation of this algorithm are:

- *m* , an integer number indicating the number of allocated processing nodes.
- *Procs[m]* , an array containing integer identifiers of the different PVM processes .
- *l[m]* , an array of integer numbers. An element *l[i]* of this array indicates the sum of the size of tasks assigned to the process identified by *Procs[i]* .
- *offset[p]* , an index sorting the tasks in downward order. *offset[0]* identifies the largest task in the task pool.

Figure 36 shows the computational load distribution, obtained with this algorithm, on two processors. Processing node 1 (star point) runs the master process and processing node 2 (circle point) runs the slave process. The different parameter values for this run were :
p = 100 ; *g* = 100 ; *n* = 1000; *d* = 6 . At each generation , the graph shows the sum complexity

of genetic programs (computational load) evaluated by the master process compared with the computational load of the slave process . We remark that the two curves are very similar.

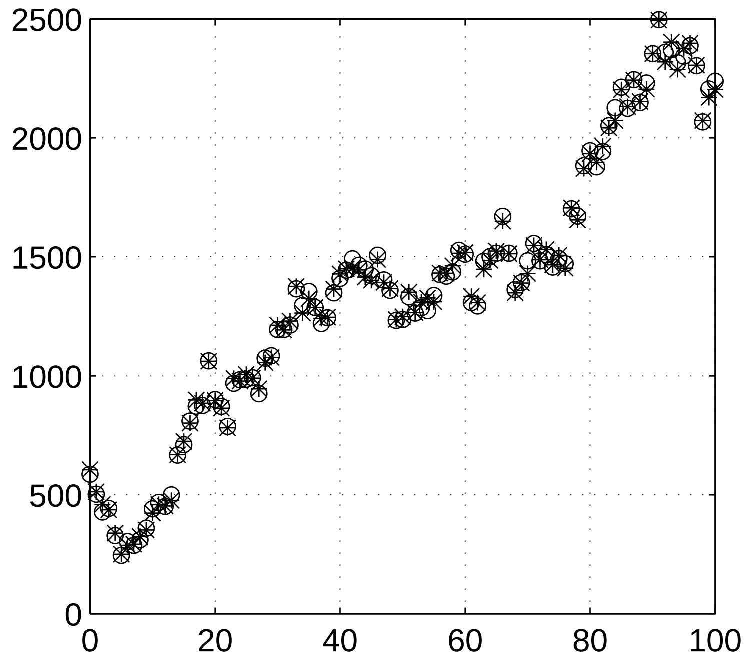


fig 36: Load balancing , by generation, on 2 processors. The star point line is the computational load curve of the master process , and the circle point line indicates the computational load curve of the slave process .

4.3 Speedup and Efficiency

The speedup ratio S is defined as

$$S_m = T_S / T_m$$

where T_S is the execution time on a single processor and T_m corresponds to execution time on m processors.

Let f be the sequential part of the program (percentage of operations of a sequential program that cannot be carried out in parallel, but must instead be executed sequentially).

The speedup ratio on *m* processors comes to

$$S_m = \frac{T_s}{fT_s + (1-f)(T_s/m)} = \frac{1}{f + (1-f)/m}$$

known as the traditional form of *Amdahl*'s law.

The fundamental limits of the speedup can be computed by increasing the number of processor nodes , m, to infinite :

$$S_m\big|_{m \to \infty} = 1/f$$

According to the formula above , the speedup S is bounded by the inverse of the sequential fraction of the application .

When computing speedups in the usual way , we consider a fixed-size problem and evaluate the ratio S of the time T_S spent by a single processor over the time T_m spent with m processors.

Another important measure of the quality of parallel applications is the efficiency E_m, defined as the ratio of the speedup over the number of processors :

$$E_m = (S_m/m) \leq 1$$

The efficiency E_m measures the average utilization of each processor .
We use both speedup and efficiency metrics for evaluating the performance of PGPS.

4.4 Scalability analysis

In this analysis, we model the distributed memory machine as a set of high-performance sequential machines interacting through a low-latency high-bandwidth network. Interprocessor communication is performed using explicit message passing.

In state-of-the-art distributed memory machines, the time to send a message containing s units of data from a processor to another processor can be modelled as $\tau_s + s\tau_d$ communication time, where τ_s is the startup time and τ_d is the data transmission rate.

In the current generation of interconnection networks, for most commercially available machines, the effects of network link contention and the distance between processors are relatively small compared with large software overheads in message passing. The statup time, including software and communication protocol overheads, is associated with each communication operation.

We define the following parameters for the purpose of analysis.

m : the number of processors M_i $(i = 1, \ldots, m)$;

p : the number of tasks T_j $(j = 1, \ldots, p)$;

c_j : the complexity of the genetic program corresponding to task T_j ;

$c_{max} = max(c_j \big| (1 \leq j \leq p))$: the size of the longest task ;

$c^* = (1/p) \sum_{j=1}^{p} c_j$:the task average size ;

l_i : the work load of processor M_i ;

n : the problem size (the number of fitness cases related to the problem to solve) ;

$l_{max} = max(l_i \mid (1 \le i \le m))$: the maximum work load assigned to a processing node at evaluation phase.

The evaluation phase can be performed in αpc^*n time on a sequential machine (see formula 4.2).

On m processing nodes, using the dynamic scheduling algorithm, the work load l_{max} assigned to the most loaded processing node can not exceed l_{min}, the work load of the least loaded processor, by an amount larger than c_{max}. If this were not the case, the excess of work $l_{max} - l_{min}$ would be composed of at least two tasks. Due to our load balancing strategy, one of these tasks could have been assigned to the least loaded processor. Thus, $(l_{max} - l_{min}) \le c_{max}$.

Since l_{min} is certainly smaller than the average load per processor $(pc^*)/m$, we have the inequality

$$l_{max} \le \left[(pc^*)/m + c_{max} \right]$$

When the population size, p, is large enough, we may then estimate that

$$l_{max} \approx (pc^*)/m$$

Since each GP is evaluated for n fitness cases, the evaluation phase will take

$$T_{comp} = (\alpha n/m) pc^*$$

where α is some constant.

Communication should also be taken into account. A genetic program of complexity c is packed as a message of length smaller than $(6c + 4)$ bytes, due to parenthesis and extra characters in the S-expression representation. The length of a message transmitted between any pair of processors is consequently less than

$$(6c_{max} + 4)$$

The total communication time at evaluation phase is bounded by the time required to move the whole population, involving one-to-all personalized communication operation with message length variance. The total length of all p outgoing messages (initiated from the master process) is $p(6c^* + 4)$ requiring a communication time which can be written

$$T_{com} = p\tau_s + p(6c^* + 4)\tau_d$$

When c^* is large, one can drop out the term $p\tau_s$ and then estimate $T_{com} = \beta pc^*$, where β is some constant.

The total execution time, t, of the evaluation phase can be modelled as linear combination of computation and communication costs:

$$t = T_{comp} + T_{com} = (\alpha n/m)pc^* + \beta pc^* \quad \textbf{(4.3)}$$

or

$$t/(pc^*) = \alpha n/m + \beta \ . \quad \textbf{(4.4)}$$

The quantities α and β can be determined from actual runs on a parallel machine.

Table 1 reports different measurements of $t/(pc^*)$ as function of n/m while varying n in $[2,2000]$ and m in $\{2, 3, 4\}$.

Figure 37 fits the set of points $(t/(pc^*) \ , \ n/m)$ as a linear function, in agreement with the model given by equation 4.4, deriving on the IBM SP-2 machine the values:

$$\alpha = 1,45\times10^{-6}$$

$$\beta = 16,70\times10^{-6}$$

These quantities are given in the unit of *sec* and are inversely proportional to an effective Mflops rate and bandwidth, respectively.

From relation (4.3), the speedup on *m* processors can be approximated as

$$S_m(n) = (\alpha nm)/(\alpha n + \beta m)$$

and the efficiency as

$$E_m = \frac{S_m(n)}{m} = (\alpha n)/(\alpha n + \beta m) \quad \textbf{(4.5)}$$

For a fixed problem size, n, the speedup saturates at value $\alpha n/\beta$ when increasing to infinite the number of processors m.
That is to say, given any number of processors m, nearly linear speedup can be obtained by simply taking a big enough instance of the problem. However, the scalability of a system is not characterized only by the speedup curve but also by the ease with which speedups are

acheived in relation with m.

A parallel system is said to be scalable if there is an isoefficiency function I such that $n = I(m)$ indicating how the problem size n must grow in order to maintain the efficiency E at a desired value while increasing the number of processors m.

From expression (4.5), we obtain

$$n = I(m) = \left(\frac{E}{1-E}\right)\left(\frac{\beta}{\alpha}\right)m$$

The isoefficiency function I is linear with respect to m, making the system linearly scalable when the population size p and the average complexity c^* are large enough.

Thus, an efficiency of $E = 0.9$ on 10 processors requires a problem size of $n \approx 10^3$ and the same efficiency can be maintained on 100 processors by increasing n to 10^4.

n	*m*	*n / m*	*p*	*g*	*t/pc**($\times 10^{-5}$)
2	2	1	100	100	2.25906
4	2	2	100	100	2.53164
10	2	5	100	100	3.7797
100	4	25	100	100	5.698
100	3	33	100	100	6.62254
100	2	50	100	100	9.99998
1000	4	250	100	100	41.2186
1000	3	333	100	100	44.9899
1000	2	500	100	100	77.54
2400	4	600	100	100	79.9253
2000	3	666	100	100	92.1774
2400	3	800	100	100	120.939
2000	2	1000	100	100	152.72

table 1: t/pc in relation with n/m.*

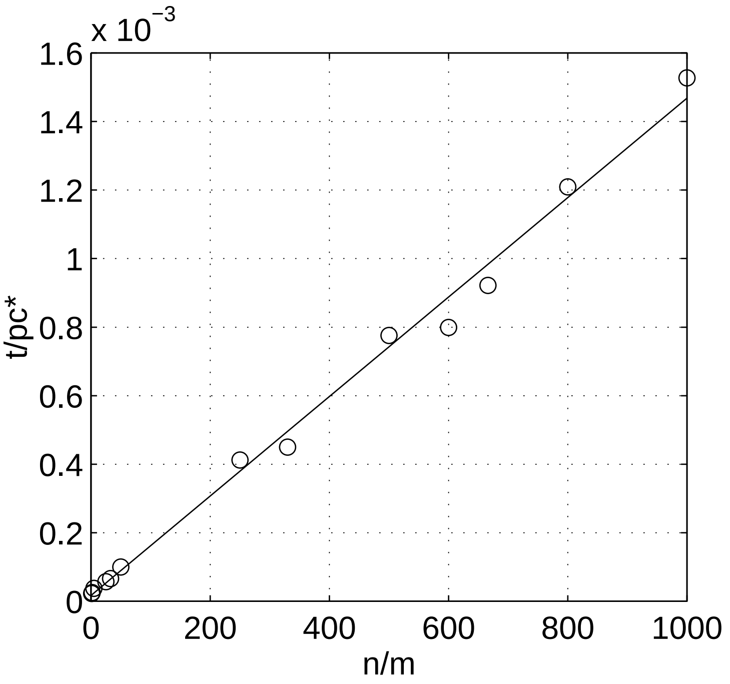


fig 37: Execution time model

4.5 Performance evaluation

The parallel genetic programming system was implemented on an SP-2 machine (described in section 4.6). We used from 1 to 10 *Thin* processing nodes to conduct our experiments.

The logistic function is used as benchmark application for the speed-up performance measurements [4]. We used a problem size (number of fitness cases) of $n = 1000$ and the speed-ups obtained are consistent with the measurements obtained for the trading model application described in chapter 5.

The traditional form of the speedup metric requires that the output of a parallel run must be the same as the one of its corresponding sequential execution. For this, we have fixed not only the problem size but also the seed for the random number generator as parameter of an evolutionary process in genetic programming. Consequently, at each generation of the evolutionary process the two runs will evolve the same population. Other authors suggest fixing only the problem size and to average over many runs [3].

The speedup was measured on two different instances, (A) and (B), of the problem and for each instance we report both static and dynamic load balancing performances. Table 2 summarizes the parameters used for each problem size. The columns *n*, *d*, *p*, *g* indicate respectively the probleme size, maximum depth, population size and number of generations. The speedups of both static and dynamic scheduling algorithms are summarized in table 3 and table 4 and the corresponding curves are plotted in figure 38 and figure 39. The elapsed time was measured in the dedicated mode.

	n	d	p	g
(A)	1000	6	100	100
(B)	100	12	100	10

table 2: A summary of the parameters for the two problem sizes

# processors	p , g , n	d	Time(s)	Speedup	Efficiency
1	100, 100, 1000	6	234.97	1	1
2	“	“	125.24	1.88	0.94
3	“	“	86.87	2.70	0.90
4	“	“	69.38	3.39	0.85
5	“	“	57.17	4.11	0.82
6	“	“	49.25	4.77	0.79
7	“	“	43.93	5.35	0.76
8	“	“	39.71	5.92	0.74
9	“	“	37.18	6.32	0.70
10	“	“	34.05	6.90	0.69
1	100, 100, 1000	6	234.97	1	1
2	“	“	121.02	1.94	0.97
3	“	“	80.75	2.91	0.97
4	“	“	63.99	3.67	0.92
5	“	“	52.38	4.49	0.90
6	“	“	44.94	5.23	0.87
7	“	“	39.56	5.94	0.85
8	“	“	35.47	6.62	0.83
9	“	“	32.15	7.31	0.81
10	“	“	29.57	7.95	0.79

Table 3: Speedups obtained using static and dynamic algorithms for $p = 100$, $g = 100$, $n = 1000$ *and* $d = 6$

For $n = 1000$ and $d = 6$, the sequential execution takes 234.97 seconds on a single node of SP-2. Using 10 SP-2 nodes, the static load balancing scheme completes in 34.05 seconds whereas the dynamic load balancing algorithm takes 29.57 seconds.

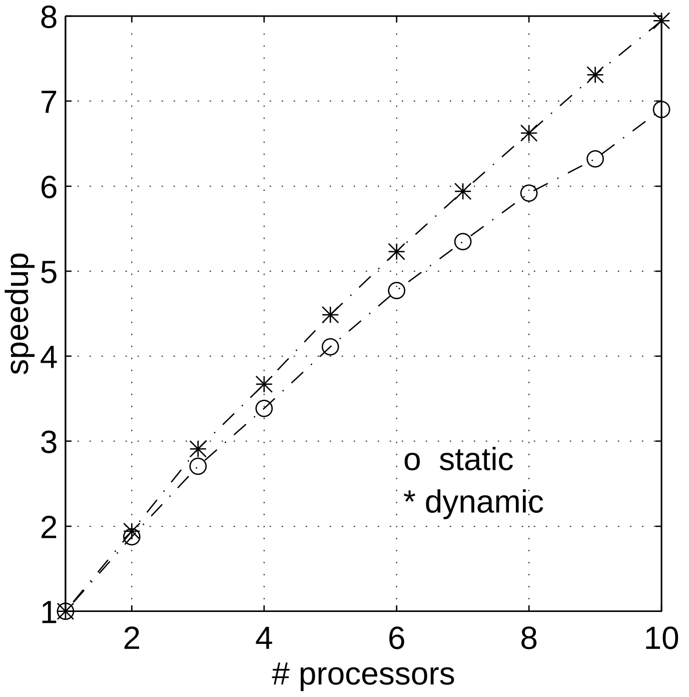


fig 38: Speedup of PGPS for $n = 1000$

When reducing the problem size to $n = 100$ and increasing the maximum depth up to $d = 12$, the sequential execution time takes 743.77 seconds on a single node. On 10 nodes, the static scheduling algorithm completes in 140.2 seconds while the dynamic one takes 105.68 seconds.

The difference in performance between the static and dynamic load balancing scheme can enlightened by the following discussion. Increasing d expands in exponential way (see formula 4.1) the reachable maximum task size and therefore augments the task average size, c^*, which scales the whole processing time.

As previously described in section 4.4, using the dynamic load balancing algorithm, the speedup saturates at $\alpha n / \beta$ when increasing the number of processors, m, to infinite. Hence, the speedup is bounded by 86.8 for the problem instance (A) and by 8.68 for the problem instance (B).

Using the static load balancing scheme, each node will receive at most $\lceil p/m \rceil + 1$ tasks to be evaluated. In the worst case, the longest tasks are assigned to the same node, thereby the work load can be bounded by $(\lceil p/m \rceil + 1) c_{max} n$.

Following the same reasoning as in section 4.4, we derive, for the static load balancing algorithm, the limit $(\alpha / \beta) (c^* / c_{max}) n$ of the speedup when m goes to infinite.

Clearly, both static and dynamic load balancing schemes will lead to similar speedup performances when the ratio $q = c^* / c_{max} \approx 1$.

That is either d is small, case of problem (A), making all the tasks equal sized or there is a large enough task T_i, of complexity c_i, such that the rest of tasks in the pool can be neglected:

$$c^* \approx c_{max} = c_i \gg \sum_{j=1}^{p} c_j \quad (j \neq i)$$

In the other case $(q < 1)$, the dynamic algorithm gives better results than the static one and this explains, in problem (B), the speedups of 5.3 and 7 obtained respectively with the static and dynamic load balancing schemes on 10 processors.

# processors	p , g , n	d	Time(s)	Speedup	Efficiency
1	100, 10, 100	12	743.77	1	1
2	“	“	437.7	1.70	0.85
3	“	“	292.81	2.54	0.85
4	“	“	259.39	2.87	0.72
5	“	“	212.02	3.51	0.70
6	“	“	195.13	3.81	0.64
7	“	“	179.8	4.14	0.59
8	“	“	165.78	4.49	0.56
9	“	“	146.4	5.08	0.56
10	“	“	140.2	5.31	0.53
1	100, 10, 100	12	743.77	1	1
2	“	“	389.04	1.91	0.96
3	“	“	269.35	2.76	0.92
4	“	“	208.72	3.56	0.89
5	“	“	172.1	4.32	0.86
6	“	“	147.9	5.03	0.84
7	“	“	132.9	5.60	0.80
8	“	“	120.19	6.19	0.77
9	“	“	113.1	6.58	0.73
10	“	“	105.68	7.04	0.70

Table 4: Speedups obtained using static and dynamic algorithms for $p = 100, g = 10, n = 100$ *and* $d = 12$

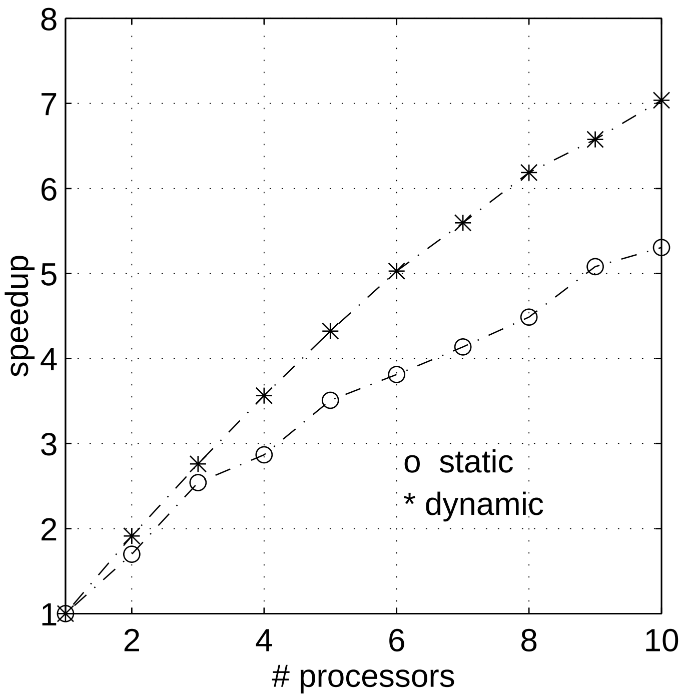


fig 39: Speedup of PGPS for $n = 100$

Note that the case where all the tasks are approximately of the same size is similar to the parallelization of the canonical genetic algorithm. The parallelization of the classical genetic algorithm solving an application (data classification) where the fitness evaluation dominates the rest of the GA calculations is reported in [5]. Sets of strings are simply sent to the processors for evaluation only. The authors have called this approach of parallelizing «*mico-grained parallelism*» and shown that this technique can produce a nearly linear speedup in GA performance.

4.6 The IBM SP-2 machine

SP-2 is a distributed memory machine using a network based computing model, providing two types of processing nodes: *Wide* and *Thin* nodes. The POWER2 processor in the *Wide* processing node runs at 66.7 MHz, giving a peak performance of 266 MFLOPS. The POWER processor in the *Thin* processing node runs at 62.5 MHz, giving a peak performance of 125 MFLOPS. The processing nodes share data via message passing over the *HPS* (*High Performance Switch*) multistage packet switched Omega network [2]. The *switch* chip provides 40 MBytes/s peak channel bandwidth and 500 *nsec* hardware latency.

Table 5 presents the performance of point-to-point communication operation using the user space implementation (dedicated nodes connected with the HPS) of IBM PVMe message passing library [1].

Size(B)	10	50	100	500	1000	5000	10000	50000	100000	500000	1000000
Time(us)	109	132	153	155	188	410	659	2023	3520	17669	35026

Table 5: Communication performance for various message sizes.

The time was measured using the wall clock *gettimeofday* in the dedicated mode. Each time reported is half the round-trip (ping-pong) communication time. In oder to minimize the effect of other user's traffic in the network, each measure is repeated 100 times and then the minimum value is retained. The code is implemented as two communicating processes (Figure 40) *process 1* and *process 2*. Note that the packing operation in step (c) is performed once only and the packing time is not included in the measurements. At each iteration in step (d), the processes are first synchronized and then the clock is started.

```
                         Process 1

(a) - prepare a message M of size S
(b) - clear the send buffer: pvm_initsend()
(c) - pack the message M into the active buffer: pvm_pkstr()
(d) - perform the following operations:
      - synchronize with process 2: pvm_barrier()
      - store the current physical time in structure t1: gettimeofday()
      - send message M to process 2: pvm_send()
      - receive synchronously a message of size S: pvm_recv()
      - compute the elapsed time in microseconds:
         elapsed_time = ( t2.tv_sec - t1.tv_sec ) * uS_PER_SECOND +
                        ( t2.tv_usec - t1.tv_usec )
(e) - repeat step (d) 100 times and take the minimum elapsed_time value: min_time
(f) - return the half ping-pong communication time: min_time / 2

                         Process 2

(a) - prepare a message M of size S
(b) - clear the send buffer: pvm_initsend()
(c) - pack the message M into the active buffer: pvm_pkstr()
(d) - perform the following operations:
      - synchronize with process 1: pvm_barrier()
      - receive synchronously a message of size S: pvm_recv()
      - send message M to process 1: pvm_send()
(e) - repeat step (d) 100 times.
```

fig 40: Code implementing two processes measuring the performance of point-to-point communication operation.

Figure 41 shows the communication performance curve. The curve shows a discontinuity at 100 bytes. For long messages (larger than 500 bytes) , the communication time becomes nearly linear as message size increases, leading to an asymptotic bandwidth of 28.5 MBytes/s.

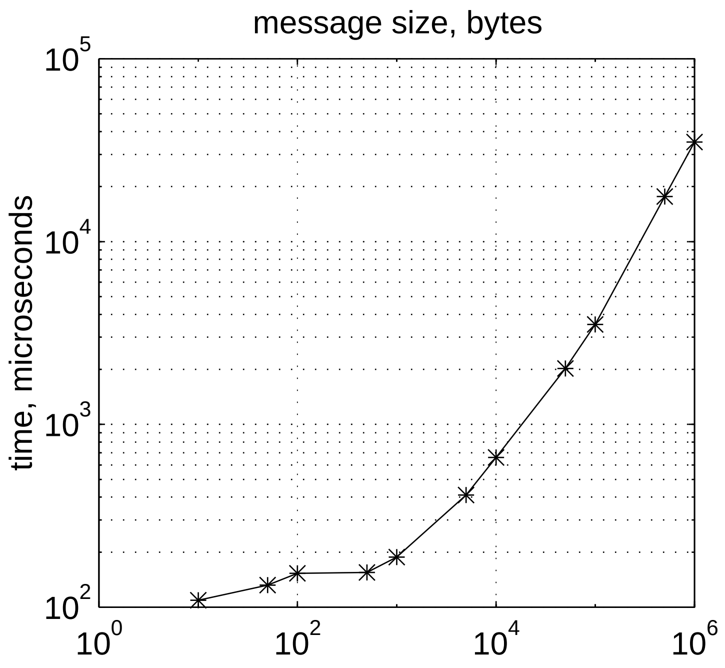


fig 41: Performance of Point-to-Point Communication Operation on SP-2

Bibliography

[1] IBM, 1994. IBM AIX PVMe User’s guide and Subroutine Reference, December 1994.

[2] IBM, 1995. IBM *System Journal*, 34(2), 1995.

[3] Koza, J. and Andre D. 1995. Parallel Genetic Programming on a Network of Transputers, Computer Science Department, Stanford University, Technical report CS-TR-95-1542.

[4] Oussaidène, M., Chopard B. and Tomassini M. 1995. Programmation évolutionniste parallèle, RenPar’7 , PIP-FPMs Mons , Belgium. Libert G., Dekeyser J.L., Manneback P. (editors).

[5] Punch, W.F., Goodman E.D., Pei M., Chia-Shun L., Hovland P. and Enbody R. 1993. Further Research on Feature Selection and Classification Using Genetic Algorithms, Appeared in ICGA93, pg 557-564, Champaign III.

5. Applying Genetic Programming to evolve financial trading models

5.1 Introduction

The question of economic forecasting is crucial in today's world. Firm, public sector, business, and consumer decisions are all based on uncertain expectations about the future.
The forecasting literature in economics can be assigned to two different frameworks. On one hand, there is a long tradition that places considerable emphasis on the theorical aspects in guiding the evaluation of econometric models. Although many authors have clearly postulated that a good forecasting performance is a necessary condition for any theory to be given such status, there is still a large number of academic economists and econometricians who tend to view the forecasting problem as one of secondary importance.
On the other hand, scientist researchers believe that understanding the structural relationships in an economic system may not be sufficient condition to forecast it well, and thus pragmatic observation rather than theorical models is the basic interest of the technical analysis.

5.2 Technical analysis

The technical approach to investment is essentially a reflection of the idea that the stock market moves in trends which are determined by the changing attitudes of investors to a variety of economic, monetary and political forces [6]. The art of technical analysis, for it is an art, is to identify changes in such trends at an early stage and to maintain an investment posture until a reversal of that trend is indicated.
Human nature remains more or less constant and tends to react to similar situations in consistent ways. By studying the nature of previous market turning points, it is possible to develop some characteristics which can help identify major market tops and bottoms.
Technical analysis is therefore based on the assumption that people will continue to make the same mistakes that they have made in the past. Human relationships are extremely complex and are never repeated in identical combinations. The stock market, which is a reflection of people in action, never repeats a performance exactly, but the recurrence of similar characteristics is sufficient to permit identifying major juncture points. As trend determining technique, we use indicators based on the price time series. Since no one indicator can ever be

expected to signal all such trend reversals, it is essential to use a number of them together so that an overall picture can be built up. We use, for this purpose, genetic programming which provides a way to formalize the trading models in a symbolic form. The real-life benchmark application addressed in this chapter shows that robust and profitable trading strategies can be inferred using genetic programming.

5.3 Trading model indicators

5.3.1 Moving Averages

The most widely used indicators in technical analysis are based on moving averages (MA) . A moving average attempts to tone down the fluctuations of stock prices into a smoothed trend, so that distortions are reduced to a minimum. We use in our study different types of moving averages.
A simple MA is constructed by totaling a set of data and dividing that total by the number of observations. In order to get the average to “move” , a new item of data is added and the first item on the list substracted. The new total is then divided by the number of observations and the process repeated ad infinitum. The number of observations is known as the range Δt_r of the MA. The choice of the range is very important. A short range MA would be so sensitive that it would continually give misleading or “whipsaw” signals. A long range MA smooths out all the fluctuations taking place during the current period. Only a MA that can catch the movement of the actual trend will provide the optimum tradeoff between lateness and oversensitivity. Here, we define a MA as the mean of multiple exponential moving averages (EMA). The advantage of crossing multiple moving averages is smoothing the data many times and thereby reducing the possibility of false signals. An EMA, as weighted moving average, has the particularity to reverse direction much more quickly than a simple MA, which is calculated by treating all the data equally. This is due to fact that an EMA gives greater weight to more recent observations. A general form of a moving average is given hereafter:

$$MA_{x,r,j,n}(t) = \frac{1}{n-j+1}\sum_{i=j}^{n} EMA_{x,r}^{(i)}(t)$$

Where : Δt_r indicates the range (the depth in the past), in days, of the moving average.

j and *n* are respectively the minimum and maximum order of the EMA operator.

$x(t)$ represents the price time series.

An *EMA* of order *i* is computed recursively by formula (5.1).

$$EMA_{x,r}^{(i)}(t) = EMA_{x,r}^{(1)}(EMA_{x,r}^{(i-1)}(t)) \quad \text{with} \quad EMA_{x,r}^{(0)}(t) = x(t) \qquad \textbf{(5.1)}$$

and

$$EMA_{x,r}^{(1)}(t) = \frac{1}{\Delta t_r} \int_{-\infty}^{t} x(t') e^{-(t-t')/\Delta t_r} \, dt'$$

Time series x(t) is a discrete time function , but can be interpolated using a linear interpolation in the time intervals between the series elements. Consequently, the formula above can be computed in recursive way as follows :

$$EMA_{x,r}^{(1)}(t) = \mu EMA_{x,r}^{(1)}(t-1) + \upsilon\,[x(t) - x(t-1)]$$

$$\text{Where} \quad \mu = e^{-\alpha} = e^{-\frac{\Delta t}{\Delta t_r}} \quad \text{and} \quad \upsilon = \frac{1-\mu}{\alpha}$$

The EMA can be initialized to the first series element :

$$EMA_{x,r}^{(1)}\ (t = 1) = x(1)$$

A part of data is reserved to build up the EMA values . This buildup period is required for reducing the initialization error , which declines in exponential way over time . The advantage of this computation scheme is that only the previous values of the EMA and price are required for evaluating the current EMA value , and thereby avoid keeping in memory a long price history .

5.3.2 Momentum

Although the indicators based on moving averages are extremely useful , they identify a change in trend after it has taken place and are helpful only when a trend reversal is detected at a relatively early stage in its development. We use here , besides moving averages , another type of indicator based on the concept of momentum. A simple form of momentum is given by :

$$m_{x,r,j,n}(t) = x(t) - MA_{x,r,j,n}(t)$$

Since the moving average represents the price trend , the resulting momentum indicator shows how fast the price is advancing or declining in relation to that trend. The momentum is null when the price reaches its MA. The momentum curve gives also useful indications of latent strength or weakness in a price trend.

Figure 42 shows the evolution of 16 days range ($\Delta t_r = 16$) moving average monitoring the USD/DEM exchange rate over one year . The minimum and maximum order of the EMA operator are respectively j = 2 and n = 4 . The moving average , abbreviated as EMA_x_16_2_4 , is represented by the dashed line . The solid line indicates the logarithmic middle price . Note that the first 150 days constitute the buildup period .

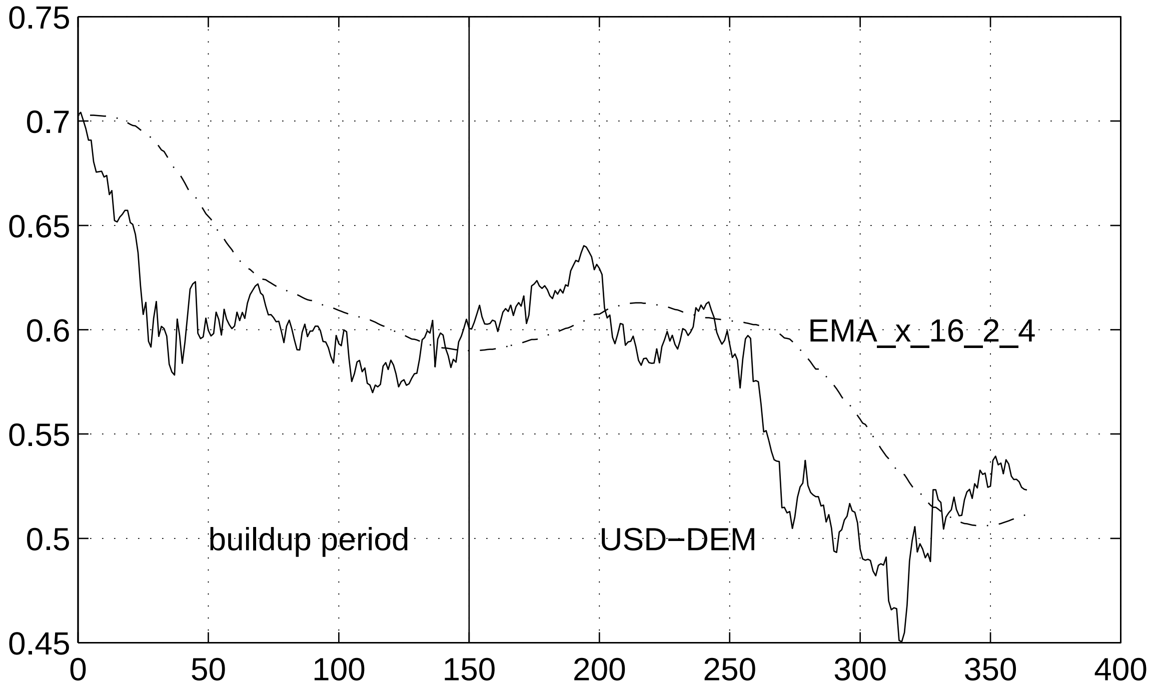


fig 42: Logarithmic evolution of the USD - DEM exchange rate over 1 year (solid line) . The dashed line indicates a 16 days moving average . The EMA order scans the interval [2 , 4] .

The deriving evolution of momentum indicator MOM_x_16_2_4 from the EMA_x_16_2_4 curve presented above is given by the figure 43. The constant (*K = 0.3*) is a threshold value, called *break-level*. Figure 42 and figure 43 run exactly the same period.

The momentum indicator is calculated as follows :

$$I_{x,r,j,n}(t) = \frac{m_{x,r,j,n}(t)}{\|m_{x,r,j,n}(t)\|} \qquad \textbf{(5.2)}$$

Since the formula (5.2) is independent of the FX rate to analyze, a scaling factor is used to normalize the value of the indicator and is given by the square root of long range momentum of the squared values of the indicator. More precisely, if *M(t)* is the time series of the squared momentum,

$$M(t) = (m_{x,r,j,n}(t))^2$$

then

$$\|m_{x,r,j,n}(t)\| = \sqrt{max\{\varepsilon, M(t) - EMA^{(1)}_{M(t),\, nr \times r}(t)\}}$$

where ε is an infinitesimal positive value , *nr* is the normalization range and *r* is the MA range used for the momentum evaluation .

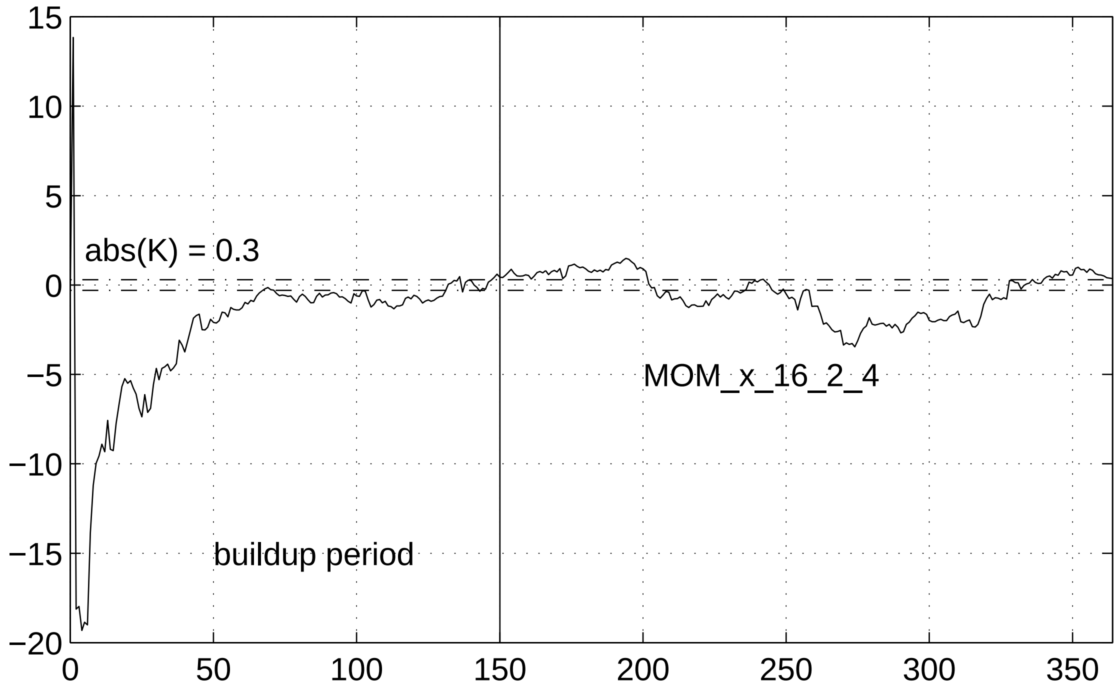


fig 43: Evolution of normalized momentum indicator "MOM_x_16_2_4" , monitoring the USD-DEM FX rate over 1 year . The break level boundaries are -0.3 and +0.3 . The interval [0,150] indicates the buildup period .

Changes in the trend of the FX rate being measured are identified not by a change in direction of the moving average , but by a crossover of the moving average and the price itself . A change from an upward trend to a downward trend is signaled when the price moves below its MA . The reverse set of conditions will confirm the termination of a downward trend . The beginning and the end of a trend development is therefore surrounded between two successive violations of the MA by the price curve .

The exploitation of the trend-determining technique described above is illustrated hereafter , assuming the analysis of the USD-DEM FX rate . If the price is above its MA ($I_x(t) > K$) then the action to take is to buy USD currency and remain in this position until the sell signal is triggered at the next crossover point . Symetrically , if the price is below its MA ($I_x(t) < -K$) then the action to take is to buy DEM currency and wait for the sell signal which will be given at the following crossover point .

The crossover region is expressed by $|I_x(t)| \le K$.
Another strategy is to compare momentum indicator with the zero reference line ($K = 0$) .

The momentum curve computed as described above, bears some fluctuations in relation to the volatility of prices . One method of filtering out these fluctuations is to smooth the momentum indicator itself by using an exponential moving average [6]. Another variation on constructing a smoothed momentum indicator is to take the momentum of a moving average (the price curve itself is first smoothed with a MA , and a momentum is taken from that smoothing) .

5.3.3 Volatility

- Another trading model indicator used in technical analysis is the *volatility.* The volatility of prices can be computed as the weighted sum of the difference between multiple exponential moving averages. A simple scheme to evaluate the volatility indicator is to combine an EMA reflecting only the current fluctuations (typically by taking a range of one hour) with EMAs with more longer range considering the price movement in the past.

Let $r_1 = 1/24$ be a range of one hour and r the range (in days) characterizing the volatility.

The normalized volatility of the price *x,* of range r , is given by

$$v_{x,\,r} = \left(\sum_{i=0}^{5} \left|dv_{i,\,x,\,r}\right| \cdot w_i \right) / \left\|v_{x,\,r}\right\| \qquad \textbf{(5.3)}$$

With $\sum_{i=0}^{5} w_i = 1$. The values $dv_{i,\,x,\,r}$ are computed as follows.

$$\begin{aligned}
dv_{0,\,x,\,r} &= EMA^{(1)}_{x,\,r_1} - EMA^{(1)}_{x,\,r} \\
dv_{1,\,x,\,r} &= EMA^{(1)}_{x,\,r_1} - EMA^{(2)}_{x,\,r} \\
dv_{2,\,x,\,r} &= EMA^{(1)}_{x,\,r_1} - EMA^{(3)}_{x,\,r} \\
dv_{3,\,x,\,r} &= EMA^{(1)}_{x,\,r} - EMA^{(2)}_{x,\,r} \\
dv_{4,\,x,\,r} &= EMA^{(1)}_{x,\,r} - EMA^{(3)}_{x,\,r} \\
dv_{5,\,x,\,r} &= EMA^{(2)}_{x,\,r} - EMA^{(3)}_{x,\,r}
\end{aligned}$$

Where the values $EMA^{(i)}_{x,\,r}$ are computed according to formula (5.1). The normalization factor represents the momentum of the variable $V_{x,\,r}$ and is computed as follows.

$$\left\|V_{x,\,r}\right\| = V_{x,\,r} - EMA^{(1)}_{v,\,nr} \quad ,$$

where the normalization range is given by $nr = 50 \times r$.

5.4 Trading models

A trading model is a system of rules catching the movement of the market and providing explicit trading recommendations for financial assets. A simple form of a trading rule could be

```
IF |I| > K THEN G := SIGN( I )                    (5.4)
ELSE G := 0
```

Where I is an indicator whose sign and value model the current trend and K is the *break-level* constant. The gearing, G, is the recommended position of the model. The value G = +1 corresponds to a ‘buy signal‘, G = -1 corresponds to ‘sell signal‘ and G = 0 corresponds to the neutral position. A trading rule with more complex strategy may use more than one indicator : typically, an indicator giving the current trend can be used in conjunction with an indicator reflecting the volatility of prices.

If $g(t)$ is the gearing position at time t , then it is updated according to table 6.

Condition	Action	Meaning
$I_x(t) > K$	$g(t) = +1$	raise a buy signal
$I_x(t) < -K$	$g(t) = -1$	raise a sell signal
$\lvert I_x(t)\rvert \le K$	$g(t) = 0$	raise a neutral signal

Table 6: Trading signals based on the indicator value.

A deal signal is given only if there is a trend reversal: $|\Delta g(t)| = |g(t_i) - g(t_{i-1})| > 0$.
Before any action to take , a trading recommendation must satisfy some timing constraints such as the local market is open and the previous deal occured at least fifteen minutes ago . The current price is checked for both validity (suitable for dealing) and reaching the stop-loss limit (the stop-loss deal turns the current position to neutral) .

The trading strategy specifies the position to be taken at the following price event , given the current gearing position and the trading signal . Figure 44 illustrates the different gearing transitions (transactions) presented as a finite state machine .

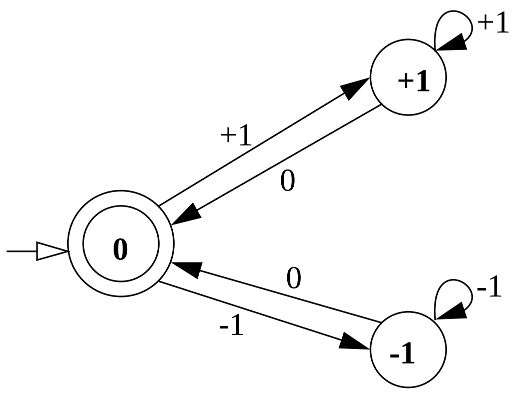


fig 44: The trading strategy .

The states 0 , +1 , -1 are the possible gearing positions . The initial and final states are given by the neutral position . A transaction holds each time a new signal is generated. Table 7 summarizes the real-time working of the automaton . The columns $sig(t_i)$, $g(t_i)$ and $r(t_i)$ indicate respectively the signal value , gearing position and deal return at time t_i .

$g(t_{i-1})$	$sig(t_i)$	$g(t_i)$	$r(t_i)$
0	+1	+1	0
1	0	0	$\left(p_{t_i}/p_{t_{i-1}}\right)-1$
0	-1	-1	0
-1	0	0	$1-\left(p_{t_i}/p_{t_{i-1}}\right)$

Table 7: The transactions and their corresponding returns .

The return $r(t_i)$ of a deal occured at time t_i is computed as follows :

$$r(t_i) \ = \ g(t_{i-1}) \ \left[\frac{p_{t_i}-p_{t_{i-1}}}{p_{t_{i-1}}}\right]$$

where $g(t_{i-1}) \neq g(t_i)$ indicates the previous gearing position and $p_{t_{i-1}}$ the transaction price of the previous deal . The transition $(1 \to -1)$ can be decomposed as sequence of $\{1 \to 0 \ ; \ 0 \to -1\}$ and vice versa . In all the other cases , the current state is preserved .

Note that the return of transactions started from the neutral position is null.

5.5 Evolving trading models

Besides of alleviating the restrictions of fixed-length representation of genetic structures, genetic programming provides a natural way to represent and evolve decision trees [1]. This section describes a way to use genetic programming to learn technical trading models for foreign exchange (FX) market. The recommendations given by a trading model are purely based on past prices (price time series) of the exchange rate being analyzed. The price history is summarized in indicators.

The trading models used in this study are combinations of rules having the form (5.4). These rules are combined by logical operators to form a decision tree [2]. The decision trees are then evolved using genetic programming where each genetic program represents a trading model.

An indicator is function of time series and, in particular, all the indicators used in this study are functions of time and price history and are based on the concept of momentum computed according to formula (5.2).

As momentum is function of the price element, its value is updated at each price event. Once updated, the normalized indicator value $I_{x,r,j,n}(t)$ is used, according to rule (5.4) , to obtain the indicator signal

$$G(x, r, j, n, K) = G(I_{x,r,j,n}(t), K)$$

The different signals returned by the indicators are then embedded in the logical S-expression corresponding to the trading model. The value obtained by the evaluation of the S-expression represents the trading model recommendation.

For illustration, the evaluation of the genetic program $(OR\quad G(x, 32, 2, 4, K)\quad G(x, 10, 2, 8, K))$ is performed as follows :

```
FOR for each price event x(t) DO
BEGIN
    (a) - update the indicator $I_{x,32,2,4}$ for the current price $x(t)$
    (b) - update the indicator $I_{x,10,2,8}$ for the current price $x(t)$
    (c) - evaluate the indicator signal $G(x, 32, 2, 4, K)$
    (d) - evaluate the indicator signal $G(x, 10, 2, 8, K)$
    (e) - perform the OR operation between (c) and (d)
    (f) - raise the deal signal found in (e)
END
```

Because of the presence of three possible values (-1, 0, +1) for the gearing signal, we have rewritten the basic logical operators:

-The OR operator returns the sign of the sum of its arguments;
-The NOT function returns the opposite decision of the argument;
-The AND function returns the neutral signal when one of its arguments is zero; otherwise returns the OR value;
-The IF function takes three arguments. It returns the second argument if the first one is true; otherwise it returns the third argument.

More formally, these logical operators can be expressed as mathematical functions where each variable a_i corresponds to an argument:

$$(OR\ a_1\ a_2) = sign(a_1 + a_2)$$

$$(NOT\ a) = -a$$

$$(AND\ a_1\ a_2) = |a_1 \times a_2| \cdot sign(a_1 + a_2)$$

$$(IF\ a_1\ a_2\ a_3) = |a_1| \times a_2 + ||a_1| - 1| \times a_3$$

Table 8, table 9 and table 10 show the truth tables corresponding to each of these logical functions.

a_1	a_2	$(OR\ a_1\ a_2)$	$(AND\ a_1\ a_2)$
-1	-1	-1	0
-1	0	-1	0
-1	+1	0	0
0	-1	-1	0
0	0	0	0
0	+1	+1	0
+1	-1	0	0
+1	0	+1	0
+1	+1	+1	+1

table 8: the OR and AND logical operators

a_1	a_2	a_3	$(IF\ a_1\ a_2\ a_3)$
-1	a_2	a_3	a_2
0	a_2	a_3	a_3
+1	a_2	a_3	a_2

table 9: the IF logical operator

a	$(NOT\ \ a)$
-1	+1
0	0
+1	-1

table 10: the NOT logical operator

The indicators in the *TerminalSet* are chosen according to their robustness. An indicator is said to be robust if it performs well in both learning process (in-sample) and test process (out-of-sample). The major problem in optimizing trading models is to avoid overfitting caused by the presence of noise. Overfitting means building trading models that fit a price history very well but generalize badly. The idea here, to avoid this phenomena, is to build genetic programs based on pre-optimized building blocks. These building blocks (robust indicators) were optimized using a niching genetic algorithm based on a fitness sharing scheme [5].

5.6 The fitness function

The fitness measure of a trading model, termed X_{eff} , quantifies not only the return but also the risk involved by the model [4]. It is defined as:

$$X_{eff} = \langle R_T \rangle - \frac{c}{2} \sigma_T^2$$

where the first term is the average return over N_T transactions (that generated a return) operated within a total test period T:

$$\langle R_T \rangle = \left(\sum_{i=1}^{N_T} r_i \right) / N_T$$

The return of a transaction is computed as described in table 7.

The second term represents the risk involved by the trading model. The constant c (= 0.1) is a risk aversion constant.

The variance of the return is computed as:

$$\sigma_T^2 = \left(\frac{N_T}{N_T - 1} \right) [\langle r^2 \rangle - \langle r \rangle^2]$$

The behavior $\sigma_T^2 = 0$ corresponds to a linear cumulated return curve. The measure X_{eff} depends on the size (in days) of the test period T. In order to enable comparisons between different periods, an annualization factor A_T is introduced [4]:

$$A_T = \frac{365}{T}$$

leading to *Annualized effective return* performance measure which can be computed and compared for different periods T:

$$X_{eff} = A_T \langle R_T \rangle - \frac{c}{2} A_T \sigma_T^2$$

This measure of trading performance through annualization is related to a single time horizon T and therefore still has a risk term associated with the period size. Averaging over n different time horizons T_i is a way to consider changes occurring with much longer or much shorter test periods:

$$X_{eff} = \left[\left(\sum_{i=1}^{n} w_i A_{T_i} \langle R_{T_i} \rangle\right) \Big/ \left(\sum_{i=1}^{n} w_i\right)\right] - \frac{c}{2}\left(\sum_{i=1}^{n} w_i A_{T_i} \sigma_{T_i}^2\right) \Big/ \left(\sum_{i=1}^{n} w_i\right)$$

Where the weights w_i are chosen so that short horizons are much more reflected than the long ones:

$$w_i = 1 \Big/ \left[2 + \log^2\left(\frac{T_i}{90}\right)\right]$$

The maximum weight is related to the 90 days horizon.

The construction of the time horizons T_i is performed such that the intervals $[T_i, T_{i-1}]$ $(n \geq i \geq 2)$ cover the full sample test period.

Since the variance is a measure of the stability of the return, then high effective return means also high stable return. The notion of robustness is directly related to the ability of generalizing the results beyond the training sample. For this purpose, each trading model is tested on more than one exchange rate time series. The fitness measure is then extended as follows:

$$X_{eff} = \langle X_{eff} \rangle - \frac{\sqrt{\sigma_{eff}^2}}{3}$$

where the first term is the fitness average value obtained for N_{FX} exchange rates:

$$\langle X_{eff} \rangle = \left(\sum_{i=1}^{N_{FX}} X_{eff,\,i}\right) \Big/ N_{FX}$$

and the second term is the variance of these values:

$$\sigma^2 = \left[\sum_{i=1}^{N_{FX}} (\langle X_{eff} \rangle - X_{eff,\,i})^2\right] \Big/ (N_{FX} - 1)$$

For this problem, the number of fitness cases is given by the size of the time series to be learned and is typically of the order of $24 \cdot 10^4$. To give a magnitude order of the time spent in the evaluation phase, evolving a population of 100 GPs with setting the maximum depth as $d = 6$, over 100 generations, takes more than 30 hours on 4 SP-2 nodes.

5.7 Performance analysis

The optimization of the trading models is performed on seven exchange rates (*GBP/USD , USD/DEM , USD/ITL , USD/JPY , USD/CHF , USD/FRF , USD/NLG*) where each time series contains hourly data and is divided into alternated periods (in-sample / out-of sample) of one and half year. The optimization period starts (*January 1, 1987*) and ends (*June 30, 1994*). The *breaklevel* was *K = 0.32* and the *buildup* period was *16*120* days.

- In a first phase, the *FunctionSet* used in designing this application is restricted to the logical functions

$$F = \{AND\ ,\ OR\ ,\ NOT\ ,\ IF\ \}$$

and the *TerminalSet* contains three pre-optimized indicators and two constant values

$$T = \{G(x, 10, 2, 9, 0.32)\ ,\ G(x, 16, 3, 3, 0.32)\ ,\ G(x, 16, 2, 4, 0.32)\ ,\ +1\ ,\ -1\}.$$

The GP evolutionary process was trained in the following manner. We performed ten in-sample runs evolving a population of 100 individuals over 100 generations (each run takes more than 30 hours on 4 SP-2 nodes). The best individuals issued from each in-sample run are then tested separately on the out-of-sample data. We report some best trading models which appears more systematically over the ten runs. The logical S-expressions are presented hereafter.

(OR (AND 1 G(*x, 16, 3, 3, 0.32*)) (AND G(*x, 16, 2, 4, 0.32*) G(*x, 16, 3, 3, 0.32*))) **(1)**

(IF (OR *G(x, 10, 2, 9, 0.32)* 1) (AND G(*x, 16, 3, 3, 0.32*) 1) G(*x, 16, 3, 3, 0.3*2)) **(2)**

The average yearly return $\langle \bar{R} \rangle$ and the fitness value X_{eff} on seven exchange rates (*BP/USD , USD/DEM , USD/ITL , USD/JPY , USD/CHF , USD/FRF , USD/NLG*) in both training and validation periods for these trading models are presented in table 11, table 12, table 13 and table 14. The column F represents the deal frequency expressed as the average number of transactions per week over the test period. The column NF (Neutral Frequency) indicates the percentage of time spent by the model in the neutral position (out-of-market) during the test period.

5.7.1 Performance of Trading model (1)

FX rate	$\langle R \rangle$	X_{eff}	F	NF
usd / dem	5.65%	2.43	1.7	26.1%
gbp / usd	6.64%	1.64	1.4	28.6%
usd / jpy	5.98%	-0.15	1.5	25.8%
usd/ itl	4.38%	0.33	1.9	26.6%
usd / chf	6.73%	1.91	1.8	25.8%
usd / frf	4.77%	1.50	1.8	26.1%
usd / nlg	4.55%	1.50	1.9	26.2%
Average	5.53%	1.31	1.7	26.5%
Fitness	1.00			

table 11: In-Sample performance of Trading model (1) . The average yearly return (in percent) and fitness value obtained on seven exchange rates are presented.

FX rate	$\langle R \rangle$	X_{eff}	F	NF
usd / dem	7.07%	-1.37	1.8	21.8%
gbp / usd	4.61%	-12.00	2.2	21.2%
usd / jpy	2.9%	-4.61	1.6	29.6%
usd/ itl	8.07%	2.55	1.5	18.6%
usd / chf	3.45%	-8.50	2.1	19.5%
usd / frf	5.71%	-1.76	1.8	21.6%
usd / nlg	6.75%	0.08	2.1	22.4%
Average	5.51%	-3.66	1.9	22.1%
Fitness	-5.35			

table 12: Out-of-Sample performance of Trading model (1) . The average yearly return (in percent) and fitness value obtained on seven exchange rates are presented.

5.7.2 Performance of Trading model (2)

FX rate	$\langle R\rangle$	X_{eff}	F	NF
usd / dem	5.70%	2.41	1.6	27.5%
gbp / usd	5.54%	0.00	1.6	30.0%
usd / jpy	5.40%	-0.95	1.6	26.8%
usd/ itl	4.45%	0.29	1.8	27.2%
usd / chf	6.37%	1.32	1.9	26.9%
usd / frf	4.15%	0.40	1.9	26.9%
usd / nlg	4.56%	1.66	1.7	27.4%
Average	5.17%	0.73	1.7	27.5%
Fitness	0.35			

table 13: In-Sample performance of Trading model (2) . The average yearly return (in percent) and fitness value obtained on seven exchange rates are presented.

FX rate	$\langle R\rangle$	X_{eff}	F	NF
usd / dem	6.02%	-2.78	1.8	23.0%
gbp / usd	4.76%	-11.71	2.2	21.8%
usd / jpy	3.09%	-2.77	1.5	30.4%
usd/ itl	7.08%	1.22	1.5	19.9%
usd / chf	4.86%	-6.04	1.9	20.4%
usd / frf	4.92%	-3.14	1.8	22.2%
usd / nlg	6.28%	-0.75	2.1	23.5%
Average	5.29%	-3.71	1.8	23.0%
Fitness	-5.10			

table 14: Out-of-Sample performance of Trading model (2) . The average yearly return (in percent) and fitness value obtained on seven exchange rates are presented.

It can be seen that the average yearly return seems to be stable in both learning and test periods. However, the return time series itself presents some fluctuations (reflected in the fitness value) during the out-of sample period.

In order to capture the essence of these trading models, we perform the following simplifications:

$$(1) \equiv \begin{cases} (\ AND\ \ G(x, 16, 3, 3, 0.32)\ \ G(x, 16, 2, 4, 0.32)\ ) & \text{if } \ (G(x, 16, 3, 3, 0.32) = -1) \\ G(x, 16, 3, 3, 0.32) & \text{Otherwise} \end{cases}$$

$$(2) \equiv \begin{cases} (\ AND\ \ G(x, 16, 3, 3, 0.32)\ \ G(x, 10, 2, 9, 0.32)\ ) & \text{if } \ (G(x, 16, 3, 3, 0.32) = -1) \\ G(x, 16, 3, 3, 0.32) & \text{Otherwise} \end{cases}$$

Though not identical, the S-expressions (1) and (2) follow the same trading strategy. In both GPs, the long (+1) and neutral (0) positions hinted at by the (primary) indicator $I_{x,\,16,\,3,\,3}$ are kept as the recommended position.
However, when the primary indicator raises a short (-1) position then the recommendation must be confirmed by a secondary indicator. The indicators $I_{x,\,16,\,2,\,4}$ and $I_{x,\,10,\,2,\,9}$ are used, respectively in (1) and (2), for validating short positions proposed by the primary indicator.
Note that the AND function is precisely a validating operator:

$$(and\ \ x\ \ y) = 0 \ \text{ if } \ (x \neq y)\ \ ;\ \ (and\ \ x\ \ y) = x \ \text{ if } \ (x = y)\ .$$

Figure 45 shows a sequence of trading positions taken by trading model (1) in relation with price movements. Long positions are taken when prices are growing and , in symmetrical way, the model adopts short positions in reaction to declining prices.

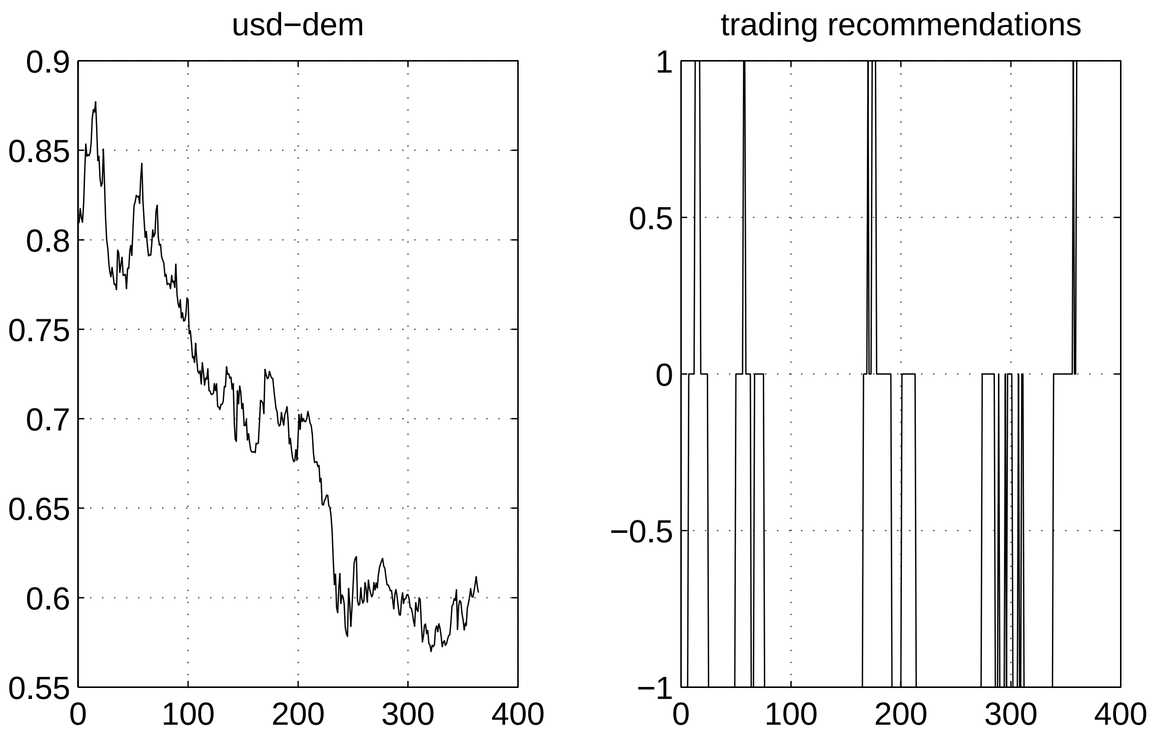


fig 45: Behaviour of trading model (1) monitoring usd-dem FX rate over one year analysis period

- Because the *FunctionSet* may be too restrictive, the search space is enlarged by including some basic mathematical functions in the *FunctionSet* [3]. The *TerminalSet* is also extended by including a volatility indicator plus real random constants in the interval [-2.0 , 2.0] .

The price volatility, termed $v_{x,\,r}$, is calculated according to formula (5.3). Also, after the evaluation of a price momentum, its value is updated according to following rule.

```
IF |I| > K THEN M := I
ELSE M := 0
```

abbreviated as

$$M(x, r, K) \;=\; M(I_{x,\,r}(t), K)$$

The real value, I, obtained by evaluating the parse tree is then used, according to formula (5.4) (with $K = 0.3$) , to raise the signal representing the recommended gearing position.

The *FunctionSet* and *TerminalSet* are composed of

$$F = \{* , / , + , - , < , > , MIN , MAX , ABS , AND , OR , NOT , IF \}$$

and the *TerminalSet* contains three pre-optimized indicators and two constant values

$$T = \{M(x, 16, 0.32) , M(x, 10, 0.28) , M(x, 8, 0.34) , V(x, 16) , RND[-2.0 , 2.0] \}.$$

The full optimization is performed in 20 independent runs. Each run evolved 4 different subpopulations, each of 100 GPs, over 100 generations. The maximum tree depth was set to 6. A migration rate of 5% GPs is done asynchronously each time a master has completed 10 generations. The imported GPs are selected according to fitness value. The same *FunctionSet* and *TerminalSet* are used by all the master nodes.

Table 15 presents the average quality of the results of the different runs compared to the average quality of the pre-optimized indicators. The results of the runs are grouped in 4 classes of decreasing out-of-sample performance.

Trading models	*Average complexity*	*In Sample*		*Out of Sample*	
		<R>	X_{eff}	*<R>*	X_{eff}
Indicators	1	5.85%	1.72	4.65%	-1.83
Run 1-5	33	5.97%	1.98	5.61%	-2.85
Run 6-10	41	6.32%	2.14	4.15%	-5.34
Run 11-15	46	6.12%	2.54	3.85%	-6.34
Run 16-20	55	8.23%	2.86	2.61%	-10.34

table 15: Average complexity, average yearly return (in percent) and fitness value corresponding to the in-sample and out-of-sample periods. The results are given for pre-optimized indicators and for each group of results.

In the first group of results (runs 1-5), the average yearly return bears some stability in both learning and test periods. However, some variability of the return appears during the test period. In the other groups, the increase of the in-sample quality is paid by a clear decrease of the out-of-sample performance, which show a higher degree of overfitting.

The best selected model in the first group has a complexity of 36 and is given by the S-expression:

(IF (MIN (ABS (MIN M(*x, 16, 0.32*) M(*x, 10, 0.28*))) (> -0.74 v(*x, 16*)))
(- M(*x, 16, 0.32*) (MIN v(*x, 16*) M(*x, 16, 0.32*)))
(- (ABS (- (ABS (MAX 0.74 v (*x, 16*)))
(> (> -0.74 v(*x, 16*)) (MIN v(*x, 16*) M(*x, 16, 0.32*)))))
(> (IF 0.3 M(*x, 16, 0.32*) v(*x, 16*)) (MIN v(*x, 16*) M(*x, 16, 0.32*))))
)

The in sample average return is 8.98% and the out-of-sample one is 5.12%. Such decision tree is not easy to interpret and some of the branches are duplicated. The solutions provided by the other runs are also complex and some simplifications must be performed to understand the information carried by a model.

Bibliography

[1] Allen, F. and Karjalainen R. 1993. Using genetic algorithms to find technical trading rules. Technical report, Wharton school, university of Pennsylvania.

[2] Oussaidene, M., Chopard B., Pictet O. V. and Tomassini M. 1996. Parallel genetic programming: an application to trading models evolution. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996*, pages 357-362. MIT Press, 1996. Proceedings of the first Annual Conference, July 28-31, 1996, Stanford University.

[3] Oussaidene, M., Chopard B., Pictet O. V. and Tomassini M. 1996. Parallel Genetic Programming and its application to trading model induction. Submitted to the journal *Parallel Computing*, July 1996 .

[4] Pictet, O.V. , Dacorogna M.M. , Muller U. A. , Olsen R. B. and Ward J.R. 1992. Real-time trading models for foreign exchange rates , Neural Network World 6/92, 713-744.

[5] Pictet, O.V., Dacorogna M.M., Chopard B., Oussaidene M., Schirru R. and Tomassini M. 1995. Using Genetic Algorithms for Robust Optimization in Financial Applications , Neural Network World 4/95, 573-587.

[6] Pring, M. J. 1988. Technical Analysis Explained, McGraw-Hill.

6. User’s Guide and Implementation Details

- This software has been run on **Unix**-based systems. It has been tested on a cluster of Sun work stations and a parallel IBM-SP2 machine. The interprocessor communications are **PVM**-based message passing routines. The system requires a **C++** compiler. If used, the graphical visualization tool needs both **Tcl** (Tool command language) and **Tk** (X windows toolkit) to be interpreted and executed.

6.1 How to prepare PGPS for solving a problem

The system may be used in three ways:

1 - Write all of the code needed to run your specific genetic programming application within the problem.cc file. In that case, the user should not need to modify the other parts of the system.

2 - Add new features (operators, selection methods, ...) without changing the structure of the routines which call them.

3 - Freely modify the content and the structure of any of the routines in the system. This provides the greatest freedom, but also the responsibility in fixing any bugs discovered in such modified version of the code.

The files to compile and link for all types of problems are the following:

file	*Objective*
TSGA.cc	The first loaded master process - main program.
TSga.cc	The master processes - main program.
allelem.cc	Mutation operation.
cross.cc	Crossover operation.
gprand.cc , gprand.hpp	Random functions.
exit.cc	Error handling.
interface.cc	Interface between a master process and the problem.
eval.cc	Work distribution and load balancing.
compare.cc	Compares between two genetic programs.
tourn.cc	Tournament selection.
select.cc	Random selection of an individual from the population.
create.cc	Creation of a genetic program.
generate.cc	Loops through the population and performs the reproduction phase.
rungps.cc	Loops through the generations and creates output files containing average fitness, average complexity and the best individual of each generation.
loadsave.cc	Input/output operations on genetic programs.
symbreg.cc	Useful routines for packing and unpacking operations.
problem.cc	Fitness function of the problem to solve.
optmain.cc	The Slave processes - main program.
optimize.cc , optimize.hpp	Routines for parsing, building and evaluating a gp.
function.cc , function.hpp	Definition of the Function (set) class.
gene.cc , gene.hpp	Definition of the Gene class.
gp.cc , gp.hpp	Definition of the GP class.
gpv.cc , gpv.hpp	Definition of the GPVariables class.
pop.cc , pop.hpp	Definition of the Population class.
terminal.cc , terminal.hpp	Definition of the Terminal (set) class.
gpmain.hpp	Includes the main class definitions (Population, GPVariables, GP, Gene).

Note that all these files can be found via anonymous ftp to cui.unige.ch . The package is located in the directory /PUBLIC/mouloud.

The system reads the parameter file gp.ini when starting each execution. An example of this parameter file looks like:

```
Function set   : *(2), +(2), -(2), /(2);
Terminal set   : rnd(-1,1),X ;
PopulationSize : 100
Generations    : 100
CreationType   : 2
MaxCreation    : 4
MaxCrossover   : 6
ADFs           : 0
Mutation       : 4
```

A full description of these parameters is given in section 3.8. The first and second lines of this file indicate respectively the set of possible functions and the set of possible terminals which can be used in a genetic program. Note that the arity of each function must be specified in parenthesis. In this example, the function *rnd* is used to generate random numbers between -1 and +1. The terminal X serves as a formal variable whose actual values represent the set of fitness cases.
The field PopulationSize and Generations give the number of genetic programs in the population and the number of generations that PGPS will perform. The initial population can be created according to several strategies (ramped half-and-half, ramped grow, ...) depending on the value of CreationType. The possible values taken by this parameter are

0 : selection of the Variable method;
1 : selection of the Grow method;
2 : selection of the Ramped Half and Half method;
3 : selection of the Ramped Variable method;
4 : selection of the Ramped Grow method.

This choice affects the structure and space complexity of the individuals. Also, MaxCreation gives the maximum depth of the lisp-expressions in this initialization phase. On the other hand, MaxCrossover defines the maximum depth of the programs after the crossover operation. Finally, the field Mutation specifies the number of individuals that will undergo mutation at each generation.

6.2 Define your own Fitness Function

The key step to use PGPS environment is to define the fitness function *EvaluateFitness*. This function is problem dependent and must be defined in the file problem.cc.
The function EvaluateFitness is called by the system for each genetic program to be evaluated.

The example below shows a template of the file *problem.cc* for fitting the logistic function.
A set of 1000 points (*x*, *y*) of this function is used for training the system. The coordinates of each point *i* are represented by (ques[*i*] , answ[*i*]). A fitness function for this problem could be chosen so as to minimize the deviation, over the training set, between the correct values and the ones returned by a genetic program. This can expressed as follows

$$min \sum_{i=1}^{1000} [GP(ques[i]) - answ[i]]^2$$

// ***problem.cc***

The file optimize.hpp is required because it contains class definitions, global variables and prototypes of routines used by the slave process.

```
#include "optimize.hpp"
```

Here, we choose an initial value, of type FITNESS (described in section 6.6), for the time series

```
FITNESS Initial_Value = 0.2;
```

The training set will be stored in two arrays. Each point *i* in the training set is represented by (ques[*i*] , answ[*i*]). The arrays ques[] and answ[] must be of type FITNESS.

```
#define SIZE 1000
FITNESS ques[SIZE];
FITNESS answ[SIZE];
```

For practical purpose, the logistic function is written as a funtion logmap(x) taking a parameter of type FITNESS and returns a value of the same type.

```
FITNESS logmap ( FITNESS x )
{
  return ( 4 * x * ( 1 - x ) );
}
```

The function *InitializeProblem()* defines the logistic function as a set of points (*x, y*). The training set is represented by (ques[*0 ... 999*] , answ[*0 ... 999*]).

```
void InitialiseProblem()
{
  ques[0] = Initial_Value;

  for ( int i = 1; i < SIZE; i++ ) {
    ques[i] = logmap( ques[i-1] );
    answ[i-1] = ques[i];
  }
  answ[SIZE-1] = logmap(ques[SIZE-1]);
}
```

Now we define the Fitness evaluation for the problem. One runs over the 1000 points constituting the training set.

```
FITNESS EvaluateFitness( GP *pgp )
{

  FITNESS rawfitness = 0, diff = 0, GPAnswer = 0;
```

Set up global genetic program variable to use ROOT macro (defined in the header file). This macro is needed to call the function *Translate* which evaluates the GP.

```
  pgpGlobal = pgp;
  Translate = TranslateROOT;
```

The following code performs the evaluation of a genetic program pgp over the training set represented by the array ques[]. Each element ques[*i*] of the data set is extracted and assigned to the global variable *globalX*. The system variable *globalX* represents the memory allocation of the terminal *X* specified in the initialization file gp.ini. Thus, the system reads the variable *globalX* each time the terminal *X* is matched in the parse tree.

```
  for (int i = 0; i < SIZE; i++ ) {
```

Extract a data element ques[*i*]

```
    globalX = ques[i];
```

Evaluate the genetic program pgp for the value *globalX*

```
    GPAnswer = Translate( ROOT );
```

Calculate the square deviation between the genetic program and the actual answer

```
    diff = ( GPAnswer - answ[i] ) * (GPAnswer - answ[i]);
```

Add this difference to total rawfitness
```
            rawfitness += diff;
        }

        return ( - rawfitness );
    }
```

In this case, the higher the rawfitness (or accumulated differences) the lower the fitness.

6.3 Handling Functions and Terminals

The user must declare every terminal and function introduced in the parameter file gp.ini. However, some classical terminals such as X and rnd are predefined. Also, the common arithmetic functions (+ , - , * , /) are already defined for the user (see *Annexe 5*).

The terminals could be variable atoms, such as X, representing the inputs of the problem. Occasionally, a terminal is a function taking no explicit arguments.
The user must declare all the input variables in the file optimize.hpp. The following line declares two variables globalX and globalY (which must be of type FITNESS) corresponding to the terminals X and Y.

```
    FITNESS globalX, globalY;
```

All the functions must be declared in the file optimize.cc. Also functions without arguments (i.e used as terminals) can be declared in this file. All the arguments of a function must be of type Gene *. Hereafter is given a declaration of the function MODULO (%). This function receives a parameter of type Gene * and returns a value of type FITNESS.

```
    FITNESS MOD( Gene *pg)
    {
        int a = Translate( pg );
        int b = Translate( pg->pgNext);

        if ( a == 0 ) return 1;
        else return ( a%b );
    }
```

Once declared, the terminals X, Y and the function % have to be handled at evaluation of the parse tree. The evaluation of the S-expression is performed by the function *Translate* located in the file optimize.cc. This function can be easily adapted to handle new terminals and functions not already declared. This can be acheived using the following pseudo-code.

```
FITNESS Translate( Gene *pg )
BEGIN

    FITNESS tmp := 0

    - Check if the node is a numerical terminal
    IF pg -> forand <> NULL THEN
        tmp := pg -> forand
        return tmp
    ENDIF

    - Check if the node is an alphabetic terminal
    IF ( pg -> pgChild = NULL) AND ( pg->Name = "X") THEN
        tmp := globalX
        return tmp
    ENDIF

    IF ( pg -> pgChild = NULL) AND ( pg->Name = "Y") THEN
        tmp := globalY
        return tmp
    ENDIF

    - The following statements deal with function nodes. All the functions in the
    FunctionSet must be defined below.
    IF pg -> Name = "%" THEN
        tmp := MOD( pg->pgChild )
        return tmp
    ENDIF

END
```

A genetic program is evaluated as many times as there are fitness cases related to the problem. The evaluation process uses the memory representation of a GP. The process starts from the root node of the parse tree and recursively applies operator to operands. Each operator is identified by the name of a function. The number of operands (arguments) gives the arity of the function. Functions without arguments are terminal nodes. In this case, the value of the terminal is returned.

If pg is a pointer to the root of a parse tree, then the evaluation could be acheived by calling this function as *Translate(pg)*.

6.4 Receiving and processing a Genetic Program

When a slave process receives a genetic program, in prefixed form, then it executes successively the following operations in order to perform evaluation.

(**1**) - Receive a genetic program;
(**2**) - Parse the GP to generate its intermediate form;
(**3**) - Build the memory representation;
(**4**) - Evaluate the fitness of the genetic program;
(**5**) - Send back the calculated fitness value.

These operations represent a slave process and are illustrated in figure 47.

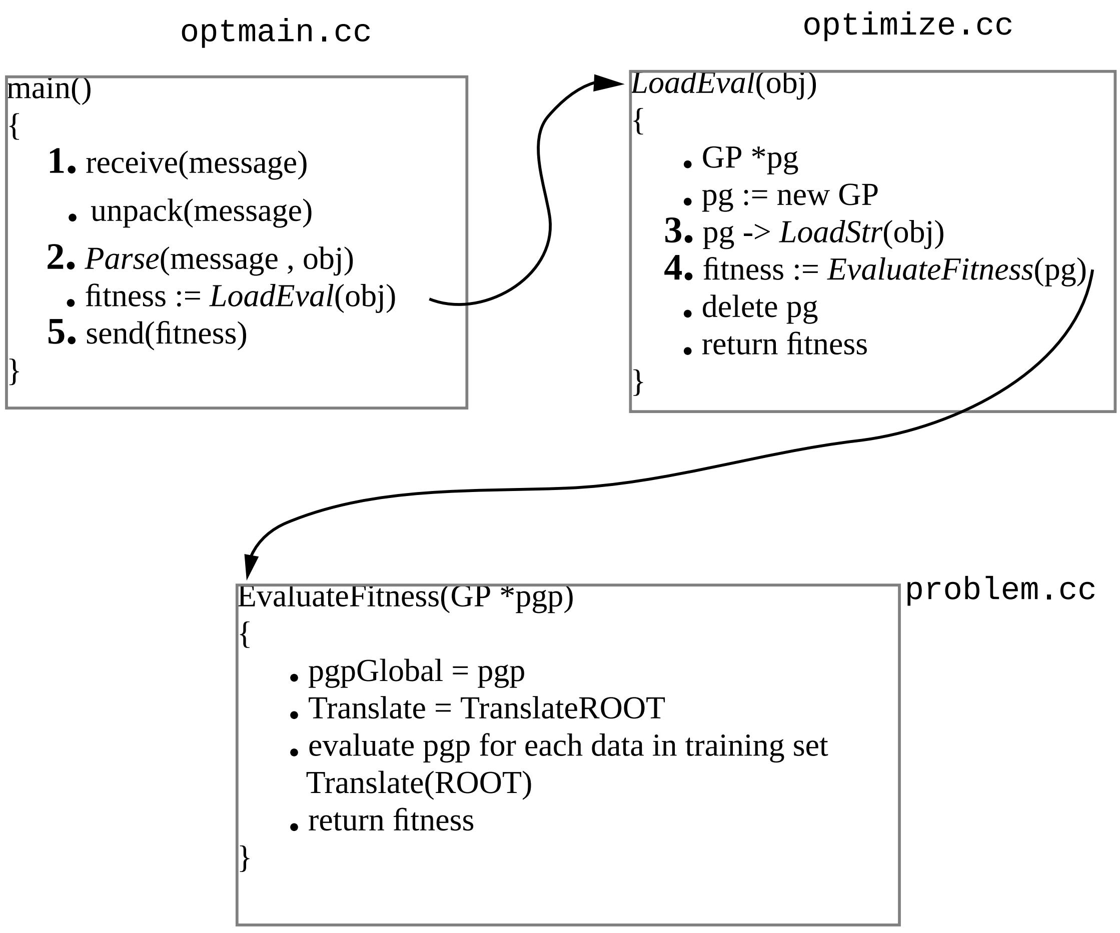


fig 46: Receiving and processing a genetic program.

The reception of a genetic program is acheived using the primitive *PVM_RECV.*
Once received, the GP is parsed to generate an intermediate form. The parser is implemented by the routine *Parse*.

After generating the intermediate form, the memory representation is built in recursive way using the routine *LoadStr.*
The GP is then evaluated by calling the function *EvaluateFitness* and the fitness value is sent back using the primitive *PVM_SEND*.

The function call *Parse*(message, obj) transforms the genetic program given by the input parameter *message* from its prefixed notation to an intermediate form which will be returned by the output parameter *obj*.

For example, the result of parsing the S-expression given by *((+ x y))* will be the character string represented by + *c x o n y o o o*

The extra-symbol c represents the child pointer, n represents the next pointer and o represents the NULL pointer.

The routine call *LoadStr*(obj, left) transforms a genetic program from its intermediate form *obj* to a memory representation (parse tree). The input parameter *left* indicates the position of the next symbol(in the buffer *obj*) to be processed. This function is described hereafter.

```
int LoadStr( buffer , left )
BEGIN

    index := left
    get( buffer, index, item)

    IF item is numeric THEN
        forand := ATONUM(item)
    ELSE
        Name := item
    ENDIF

    index := seek()
    getchar( buffer, index, ch)

    IF ch = ‘c‘ THEN
        index := seek()
        pgChild := new Gene
        index := pgChild -> LoadStr(buffer, index)
    ENDIF

    index := seek()
    getchar( buffer, index, ch)

    IF ch = ‘n‘ THEN
        index := seek()
        pgNext := new Gene
        index := pgNext -> LoadStr(buffer, index)
    ENDIF

    return index

END
```

The item at position *left* , in the buffer, is extracted. The data member *forand* is set if the item is numeric, otherwise the name of the item is stored in the data member *Name*. The function seek is used to skip white spaces. The primitives *get* and *getchar* extracts respectively an item and a character. The primitive ATONUM represents the common *atoi* or *atof C* functions.
If the current character is ‘c‘ then a new gene is allocated and the routine *LoadStr* is called in recursive way starting from the current node. The actual allocated node is a child node (the first argument) of some function. In similar way, the character ‘n‘ indicates that there is a next argument following the current one and again a new gene is allocated and a recursive call is made to follow the construction of the branch. At the end of the construction of a branch, the function returns the position, in the character string, of the last processed symbol. *Annexe 3* shows the implementation of the routines *LoadStr*, *LoadEval* and a complete example of a slave process illustrating the execution of the previous steps.

6.5 Compiling and linking

After having written the appropriate functions (initialization and fitness evaluation) for solving a specific problem, the system can be compiled using the make utility. The following file shows an example of a *makefile* that can be adapted to the user own environment.

```
MASTER = function.cc terminal.cc gene.cc gp.cc gpv.cc pop.cc allelem.cc create.cc
compare.cc eval.cc exit.cc generate.cc cross.cc rungps.cc select.cc loadsave.cc
tourn.cc gprand.cc symbreg.cc interface.cc optimize.cc problem.cc

SLAVE = optmain.cc problem.cc optimize.cc

### Use -DDOUBLE or -DINT for the fitness type .

### Use -DISLAND=value for evolving multiple populations

FLAGS =  -w -DDOUBLE -DISLAND=1 -L/usr/lpp/ssp/css/libus -lcss
PVMEXP = /usr/lpp/pvm3/lib/pvm3e.exp
LIB = pvm3
PVMDIR = /usr/lpp/pvm3/lib

all  :
    g++ -o TSGA TSGA.cc $(MASTER) $(PVMEXP) -l$(LIB) $(FLAGS) -L$(PVMDIR)
    g++ -o TSga TSga.cc $(MASTER) $(PVMEXP) -l$(LIB) $(FLAGS) -L$(PVMDIR)
    g++ -o TS $(SLAVE)  $(PVMEXP) -l$(LIB) $(FLAGS) -L$(PVMDIR)

TSGA : $(MASTER)
    g++ -o TSGA TSGA.cc $(MASTER) $(PVMEXP) -l$(LIB) $(FLAGS) -L$(PVMDIR)

TSga : $(MASTER)
    g++ -o TSga TSga.cc $(MASTER) $(PVMEXP) -l$(LIB) $(FLAGS) -L$(PVMDIR)

TS : $(SLAVE)
    g++ -o TS $(SLAVE)  $(PVMEXP) -l$(LIB) $(FLAGS) -L$(PVMDIR)

clean:
```

6.6 Macro definitions

The makefile provides macros that directs the system to use integer-valued or real-valued fitness function. When ephemeral random constants are included in the *TerminalSet* and incorporated into the initial random population, their type and range must be appropriate for the problem to solve. Ephemeral random constants might be integers, natural numbers, floating-point numbers, logical constants, etc.
If the double precision floating-point arithmetic is to be used in the evaluation of S-expressions, then the ephemeral random constants and all the constants (terminals) defined in the *TerminalSet* should be coerced into double precision.

-DDOUBLE

The -DDOUBLE macro directs the system that each run must satisfy all the constraints related to double precision handling:

- The fitness type is double;
- The system uses real random constants in the *TerminalSet*;
- The pack operation uses double conversion type to format numerical terminals;
- The unpack operation requires the *atof* function for string-to-double conversions.

This macro is implemented as described hereafter.

```
#if defined(DOUBLE)
  #define FITNESS double
  #define FORMAT "%f "
  #define ATONUM atof
  #define RANDOM(x,y,z) ( get_random_real( x , y , z ) )
#endif
```

Whenever the ephemeral random constant rnd(x,y) is chosen for any terminal of the tree during the creation of the initial random population, a real random number in the range [x , y] is generated. The precision is determined by the ratio 1/z. The argument z is by default set to 100.

-DINT

The -DINT macro directs the system that each run must satisfy all the constraints related to integer precision handling:

- The fitness type is integer;
- The system uses integer random constants in the *TerminalSet*;
- The pack operation uses integer conversion type to format numerical terminals;
- The unpack operation requires the *atoi* function for string-to-integer conversions.

This macro is implemented as described below.

```
#if defined(INT)
  #define FITNESS int
  #define FORMAT "%d "
  #define ATONUM atoi
  #define RANDOM(x,y,z) ( get_random_int( x , y ) )
#endif
```

Whenever the ephemeral random constant rnd(x,y) is chosen for any terminal of the tree during the creation of the initial random population, an integer random number in the range [x , y] is generated.

-DISLAND=value

A major capability of PGPS is to allow a number of island subpopulations running in parallel. This feature is provided using two main programs (*TSGA.cc* and *TSga.cc*) which differ only in the initialization routines. All other code is common (or compatible) between the single population and multiple subpopulations systems. The single population system is in *TSGA.cc* file and the island Parallel Genetic Programming System (*iPGPS*) uses both *TSGA.cc* and *TSga.cc* files.

iPGPS allows the user to evolve multiple subpopulations, with periodic interchange of individuals among the various subpopulations. The migration scheme between subpopulations follows the ring topology. The different migration phases related to each evolutionary process are performed asynchronously. This coarse-grain parallelism is intended to assist the user in avoiding premature convergence on difficult optimization problems.

The -DISLAND macro allows the user to specify the number of subpopulations (islands) to evolve in parallel. If this flag is not used, then a single population (-DISLAND=1) is set by default.

The number of subpopulations (*nb_islands*) to evolve must not exceed the actual number of the allocated nodes (*nhost*). At initialization, the system uses these two environment variables to configure dynamically its parallel scheme before starting any evolutionary process.

The possible configurations depends on the values of the variables *nb_islands* and *nhost* in the following way:

(1) *nb_islands* = *nhost* = 1;
(2) *nb_islands* = *nhost* > 1;
(3) *nb_islands* < *nhost*.

- In (1) , the system runs in sequential mode evolving a single population.

- In (2) , each host in the virtual machine evolves its own subpopulation. The migration phase involves all the allocated nodes.

- In (3) , the virtual machine is partitioned into two groups of processing nodes:
nb_islands nodes (*group1*) will execute evolutionary processes and
nb_slaves (= *nhost* - *nb_islands*) nodes (*group2*) will be dedicated slaves. These processing nodes will be dedicated for fitness computations. Migration of individuals holds between nodes belonging to *group1* according to ring topology.

The following pseudo code shows the implementation of the -DISLAND macro.

```
1. Initialize the number of hosts
   nhost = 1

2. Initialize the number of subpopulations to evolve
   nb_islands = 1

3. Initialize the number of slave processes
   nb_slaves = 0

4. Initialize the total number of PVM processes
   nb_procs = 1

5. Check if multiple subpopulations are to be evolved
   if defined(ISLAND) then
         nb_islands = ISLAND
   endif

6. Read the number of allocated hosts
   pvm_config(&nhost, NULL, NULL)

7. Adjust the number of islands according to nhost
   nb_slaves = nhost - nb_islands
   if nb_slaves < 0 then
         nb_islands = nhost
         nb_slaves = 0
   endif
nb_procs = nb_slaves + nb_islands
```

6.7 Tcl requirements

Tcl stands for *Tool Command Language*. Tcl provides a scripting language, and an interpreter for that language that is designed to be embedded in user's application. Tcl is similar to other Unix shell languages such as the Bourne Shell, C Shell, Korn Shell and Perl. Its associated X windows toolkit, Tk, defines Tcl commands that let create and manipulate user interface widgets. The Tcl/Tk packages run on various Unix platforms [3]. The script for the visualization tool can be found in the command.tcl file. Note that the first line of the file

```
#!/usr/local/tk4.1/bin/wish4.1 -f
```

names the interpreter for the rest of the file. Once this line is set to the actual location of *wish* program, the easiest way to use the graphical tool (gt) is to define an alias

```
gt=~mouloud/TCL/command.tcl
```

The script reads periodically the data files (*fitness* and *complexity*) produced by each run of PGPS and plots, by generation, both fitness curve corresponding to the best-of-generation genetic program and population average complexity curve.

6.8 PVM requirements

PVM stands for Parallel Virtual Machine. It is a software system that permits a network of heterogeneous UNIX computers to be used as a single large parallel computer [1]. Thus large computational problems can be solved by using the aggregate power of many computers. A user defined collection of serial, parallel and vector computers appears as one large distributed- memory computer. The term *virtual machine* is often used to designate this logical distributed- memory machine, and *host* refers to one of the member computers.

PVM supplies the functions to automatically start up tasks on the virtual machine and allows the tasks to communicate and synchronize with each other. A task is defined as a unit of computation analogous to a UNIX process. Applications, which can be written in Fortran, C or C++, can be parallelized by using message-passing constructs common to most distributed-memory machines. By sending and receiving messages, multiple tasks of an application can cooperate to solve a problem in parallel. The data conversion that may be required when two computers use different integer or floating point representations is handled.

The PVM system is composed of two parts.
The first part is a deamon process (sometimes abbreviated *pvmd*) that resides on all the computers making up the virtual machine. *Pvmd* is designed so any user can install this deamon on a machine. The user first creates the virtual machine by starting up pvmd, next starts a PVM application from a UNIX prompt on any of the hosts.
The second part of the system is a library of PVM interface routines (libpvm.a) . This library contains user callable routines for message passing, spawning processes, coordinating tasks,

and modifying the configuration of the virtual machine. Application programs must be linked with this library to use PVM.

PVM uses two environment variables when starting and running. These two variables must be set before any use of PVM. The first variable is PVM_ROOT, which is set to the location of the installed pvm directory. The second variable is PVM_ARCH, which tells PVM the architecture of the current host and thus what executables to use from the PVM_ROOT directory. This variable is automatically determined during installation.
The easiest method is to set these variables in the .cshrc file. Here is an example assuming the use of the IBM/RS6000 architecture:

```
setenv PVM_ROOT /home/pvm
setenv PVM_ARCH RS6K
```

Both pvmd and libpvm.a are placed in PVM_ROOT/lib/PVM_ARCH during compilation of the PVM package. The default location of pvmd can be overridden by setting the environment variable PVM_DPATH.

Note that PVM looks for user executables in the directory $HOME/pvm/bin/PVM_ARCH. If PVM is installed in a single location like /user/local for all users, then each user should still create $HOME/pvm/bin/PVM_ARCH to place his own executables. For example, if a PVM task called tsga is to be spawned on an IBM-SP2 machine, then on this host there should be an executable file $HOME/pvm/bin/RS6K/tsga.

The most popular method of running PVM is to start the console pvm then add hosts interactively. The PVM console, called pvm, is a stand alone PVM task which allows the user to interactively start, query and modify the virtual machine. The console may be started and stopped multiple times on any of the hosts in the virtual machine. Starting the console without any option can be done by entering the command *pvm* from unix prompt. Once started the console prints the prompt:

```
pvm>
```

Here are some useful console commands:

add	: adds hosts to the virtual machine.
conf	: lists the configuration of the virtual machine.
delete	: deletes hosts from the configuration.
halt	: kills all PVM processes (including console) and shuts down PVM.
ps -a	: lists all processes currently running on the virtual machine.
quit	: exits console leaving daemons and PVM processes running.
reset	: kills all PVM processes leaving the deamons in an idle state.
spawn	: starts a PVM process.

A C or C++ program that makes PVM calls needs to be linked with libpvm.a.
Once PVM is running, an application using PVM routines can be started from a UNIX command prompt on any of the hosts in the virtual machine.

6.9 IBM AIX PVMe

The IBM AIX PVMe program product is an implementation of PVM designed to run on SP2 machine and thus takes full advantage of the High Performance Switch of the SP2 architecture [2]. Note that PVM uses standard protocols like TCP/IP to exchange data between processors. This makes the package easily portable.

PVMe uses two exported environment variables. The first variable is PVMDPATH, which indicates the absolute path to the directory containing the PVMe deamon executable. The second variable is PVMEPATH, which tells PVMe the absolute path to the directory containing the user's executables. For instance, the following lines can be used to set these variables.

```
#!/bin/ksh
# the directory where PVMe has been installed
export PVMDPATH=/usr/lpp/pvm3

# the directory for application executables
export PVMEPATH=$HOME/Work
```

C (or C++) programs need to be linked with both pvm and css libraries. CSS (Communication SubSystem) is a low-level communication protocol that runs on the *HPS* adapters. Moreover, the user must supply the linker with an exported file pvme.exp to resolve symbols not defined in libpvm.a.

The example below shows a makefile for compiling a C++ program, called source.cc, which makes calls to PVMe (version 3) routines.

```
CC = gcc
FLAGS = -O
PVMEXP = /usr/lpp/pvm3/lib/pvm3e.exp
PVMLIB = -lpvm3
CSSLIB = -lcss

.cc.o:
        $(CC) -c $(FLAGS) $*.cc

binary: source.o
        $(CC) source.cc $(PVMEXP) $(PVMLIB) $(CSSLIB) -o binary
```

Once compiled, the user application can be spawned either from the Unix prompt or PVMe console (which allows the same commands as the PVM console). Note that the user program will run only on the set of nodes actually allocated by the Ressource Manager. This can be achieved by spawning the application from the console (on the local host) or using a remote connection (rlogin) to one of the allocated node.

The user should be sure that the deamon has started before the user application is started. The PVMe deamon accepts a parameter on the command line which specifies how many nodes are involved in the execution (this allocation request is transmitted to the Resource Manager). The following example shows an interactive session of PVMe deamon.

```
alias pvme=$PVMDPATH/pvm
pvme 2
PVMD: assuming default control workstation name
PVMD: trying to get 2 nodes
PVMD: using sp12 with dx=/usr/lpp/pvm3/pvmd3e, ep=/u/mouloud/Work
PVMD: using sp13 with dx=/usr/lpp/pvm3/pvmd3e, ep=/u/mouloud/Work
PVMD: Ready for 2 hosts...
pvm> spawn -> TSGA
1 successful
t10010000
task 10010000 on sp12: Genetic Programming System Completed.
New epoch
pvm>halt
pvm3 exiting
$
```

The following routines are not supposed to be modified but are included for reference.

6.10 Packing and Sending a Genetic Program

The function *TranslateStr* converts a terminal or a function node from its memory representation to a character string. The parameter *pg* is a gene pointer and the parameter *buffer* is character string containing the result of the conversion. The primitive *chars* converts an integer or real number to a character string. The procedure *TranslateStr* is given hereafter .

```
TranslateStr( buffer , pg )
BEGIN

    - Check if the gene is a numerical terminal
    IF pg->forand <> 0 THEN
        buffer := buffer + chars(forand)
        return
    ENDIF

    - Check if the gene is an alphabetical terminal
    IF pg->pgChild is NULL THEN
        buffer := buffer + pg->Name
        return
    ENDIF

    - Otherwise the gene is a function node
    buffer := buffer + ‘(‘
    buffer := buffer + pg->Name
    return

END
```

The function *Pack* takes a parse tree as argument and returns its underlying Lisp S-expression. This routine is useful in sending a genetic program for evaluation or saving an individual in a data file. It works as follows.

```
Pack(buffer , pg)
BEGIN

    TranslateStr(buffer , pg)
    Check if the gene has a child
    IF pg->pgChild <> NULL THEN
        Pack(buffer , pg->pgChild)
    ENDIF

    Check if the gene is followed by another operand
    IF pg->pgNext <> NULL THEN
        Pack(buffer , pg -> pgNext)
    ELSE
        buffer := buffer + ‘)‘
    ENDIF

END
```

pg is a pointer to the root node of the parse tree and *buffer* is a character string that will contain the resulting S-expression.
Packing a non terminal genetic program can be done as follows:

```
Expression := ‘(‘
Pack(Expression , pg)
```

Once packed, the GP can be sent as a character string. If *msgid* is a message identifier, and procid identifies a receiving processor then the following lines can be used to send a genetic program.

```
PVM_initsend(0)
PVM_pkstr(Expression)
PVM_send(procid , msgid)
```

A detailed description of the routines *TranslateStr* and *Pack* is given in *Annexe 2*. Also, an example illustrating the use of these routines is shown in the procedure *Evaluate*.

6.11 Crossover operator

- Crossover operator is implemented by the function *Cross*. The function takes two parental individuals P_1 and P_2 and produces a child individual P_3. The main steps executed during crossover operation are the following.

1. Copy P_1 to P_3

2. *Choose* a node n1 on genetic program P_3

3. If n1 is the root of some subtree then delete the subtree leaving the root node n1

4. *Choose* a node n2 on genetic program P_2

5. The node n1 will have the same identifier as node n2. Also, if n2 happens to be a function node then its argument must be copied. Note that n1.pgNext must be kept unchanged.

```
n1.iValue := n2.iValue
n1.forand := n2.forand
n1.pgChild := n2.pgChild
```

6. if *Depth*(P_3) > maximum allowable depth then *GOTO* 1

- The function *Choose* selects randomly a node on the parse tree. The code is written such that there will be a high probability of getting a function node. This is produced by going through a loop 10 times and returning only if a function is found. The following steps show the principle of this routine. Let *pg* be a pointer to a genetic program.

```
1. calculate the length (complexity) of the individual pointed by pg
pg -> Length( c )

2. FOR i := 1 TO 10 DO
BEGIN
    - generate a random number, j, between 1 and c
    j := RND(1 , c)

    - find the gene corresponding to value j
    pt := Nth(j)

    - if this pt points to a function then return this node
    IF pt -> pgChild THEN
        return pt
    ENDIF
ENDFOR

3. return the chosen node, even if it is a terminal.
return pt
```

- The function *Length* returns the complexity, *c*, of a genetic program. The variable *c* is incremented for each visited node. The first call to this function must be *Length*(c := 0). The pseudo-code corresponding to this routine is given below.

```
int Length (c)
BEGIN

    c := c + 1

    IF pgChild <> NULL THEN
        c := pgChild -> Length(c)
    ENDIF

    IF pgNext <> NULL THEN
        c := pgNext -> Length(c)
    ENDIF

    return c
END
```

- The function *Nth* returns the gene at position *n* on the genetic program.

```
Gene Nth(&n)
BEGIN
    pg := NULL
    n := n - 1

    IF n = 0 THEN
        return this
    ENDIF

    IF pgChild <> NULL THEN
        pg := pgChild -> Nth(n)
    ENDIF

    IF pgNext <> NULL AND pg = NULL THEN
        pg := pgNext -> Nth(n)
    ENDIF
    return pg
END
```

The parameter *n* indicates the position of a gene on the parse tree starting from the root node. Note that this argument is passed by reference so that each recursive call to this function will receive the position of the current node as parameter. The function returns a pointer to gene at position *n*. If an erroneous value of *n* is given as argument then function will return a NULL pointer.

- The function Depth returns the depth of a genetic program. The depth is calculated by taking the maximum between the depth of the left subtree and the depth of the right subtree. The following algorithm describes the depth computation. The call root -> *Depth*(0) should give the depth of the genetic program pointed by *root*. Note that the depth of a single terminal is zero.

```
int Depth (d)
BEGIN
    left := d
    right := d

    IF pgChild <> NULL THEN
        left := pgChild -> Depth (d+1)
    ENDIF

    IF pgNext <> NULL THEN
        left := pgNext -> Depth (d)
    ENDIF

    return MAX(left , right)
END
```

The implementation of the functions *Cross* , *Choose* , *Depth* , *Length* and *Nth* is shown in *Annexe 4.*

Bibliography

[1] Geist, A., Beguelin A., Dongarra J., Jiang W., Manchek R. and Sunderam V. 1994. PVM3 User’s Guide and reference manual, ORNL/TM-12187, September 1994.

[2] IBM, 1994. IBM AIX PVMe User’s guide and Subroutine Reference, December 1994.

[3] Welch, B. 1995. Practical Programming in Tcl and Tk. DRAFT, January 13, 1995.

7. Conclusions

To conclude this study of Genetic Programming, we first summarize its contents and then highlight its contributions.

This thesis discusses genetic algorithms (GAs) as adaptative methods which can be applied to search and optimization problems. The fundamentals of GAs are reviewed in chapter 1. The most important difference separating GAs from other conventional optimization techniques is that genetic methods search from a population of points (i.e parallel search) rather than a single point. By maintaining a population of well-adapted sample points, the probability of reaching a local peak is reduced.

Chapter 2 introduces progressively the concept of genetic programming as an extension of GAs to the space of computer programs.

The Parallel Genetic Programming System (PGPS) is described in Chapter 3. It is a parallel implementation of the genetic programming paradigm designed to run on distributed memory machines. The system is written in C++ using the PVM3 message passing library. It consists of 20,000 lines of code.
An application of PGPS to fit the logistic function is reported. This function is chosen for its chaotic (but deterministic) behavior as the first step to time series predictions. The results show the impact of the terminal and function sets on the quality of the inferred genetic programs.

The complexity and scalability of PGPS are analysed in chapter 4. The parallel scheme consists of multiple Master-Slave instances sharing the processing nodes. Each instance is mapped on all the allocated nodes and each master node runs an evolutionary process evolving its own subpopulation. To relieve premature convergence, the different subpopulations interact asynchronously through the ring topology. As the evaluation time is scaled by the number of fitness cases, large problem sizes make the evaluation phase compute intensive. The dynamic load balancing algorithm enhanced processor utilization by considering the task grain variance at run time, as opposed to the static algorithm which assumes equal sized tasks. At evaluation, multiple threads are maintained on each node to overlap communication with useful computation.
Given a problem size n fitness cases, the speedup presents a saturation at about $8,68\times10^{-2}n$ on

the IBM SP-2 machine. For $n = 10^2$ and $n = 10^3$, we obtained respectively the speedups of 7 and 8 over 10 processing nodes.
The isoefficiency function stating how the problem size must grow with respect to the number of processors in order to maintain the efficiency at a desired value is linear, making the parallel implementation linearly scalable. An efficiency of 0.9 on 10 processors requires a problem size of $n \approx 10^3$ and the same efficiency can be maintained on 100 processors by increasing *n* to 10^4.

Genetic programming has been demonstrated in a variety of applications, many of which have known optimal solutions determined in advance. This leaves open the question as to whether GP can ‘scale up‘ to real-world situations, where answers are not known and data is noisy.
Chapter 5 describes the application of genetic programming to infer robust financial trading strategies. Real-world price data, covering seven exchange rates (*GBP/USD, USD/DEM, USD/ITL, USD/JPY, USD/CHF, USD/FRF, USD/NLG*), is used to optimize the trading strategies. The optimization period, containing hourly data, starts *January 1, 1987* and ends *December 31, 1994*.
The average returns provided by the inferred models are robust and profitable. Typically, the average return exceeds 5%. These models can be expressed as logical combinations of robust indicators.
Our results show that when addressing such highly complex problems it becomes necessary to decompose the problem representation in such a way that an overall solution can be viewed as a combination of small modules. Genetic programming has exhibited reasonable promise for trading model optimization which is not a well understood domain.

Chapter 6 provides an understandable PGPS user‘s guide and describes methodically how to define a fitness function for a particular problem.

- An important contribution of this study has been the complexity analysis of a sequential genetic programming run in relation to the pertinent input parameters such as the population size, number of generations, maximum depth, population average complexity and the problem size (number of fitness cases).
We have observed that the population average complexity increases over time and that the evaluation phase takes most of the run-time when fitting problems requiring more than 1,000 fitness cases. Consequently, it becomes necessary to parallelize the evaluation phase when addressing real-life applications involving a large amount of data, such as time series modelling.

- The problem of task size variance arises when parallelizing the population evaluation in genetic programming. This problem is equivalent to finding an optimal schedule of *p* independent tasks on $m < p$ machines and is known to be NP-hard.
This thesis has presented a novel dynamic load balancing scheme considering the task grain variance at run time.

- We have presented an original parallel scheme of genetic programming which not only speeds up the searching process by parallelizing the evaluation phase, but also prevents premature convergence by maintaining multiple evolutionary processes with the possibility of migration between the subpopulations.

- And last, but not least, the application of genetic programming to evolve financial trading strategies yields profitable and robust trading models providing reasonable average returns. To minimize overfitting, caused by the presence of noisy data, during optimization, some precautions were considered. Each trading model is tested on more than one exchange rate, the data is separated (in-sample / out-of-sample) inside each price time series and the fitness function penalizes unstable returns.
The problem representation is an important feature in complex applications like trading model optimization. Our results show that logical combinations of robust trading rules may lead to robust and profitable trading strategies.

We think that the most important direction for future research is to explore ways in which programs constructed under genetic search can be mined for building blocks in the form of subexpressions which illustrate salient problem elements and the relationships between them. Genetic programming is likewise a symbolic method of induction, and so has potential to feed symbolic knowledge about what it has learned back into the user environment. The goal of automating the extraction of knowledge requires that information can be extracted from programs regardless of their complexity. One approach to do so, is to maintain a dynamic library along with the evolutionary process. This dynamic library can be viewed as an extension of genetic programming to address the problem of automatic subroutine generation. One can proceed by randomly extracting subexpressions from fitter programs in the population and reformatting them into parameterized functions (modules) which will be stored in the library. Thus, the modules extend the function set from which programs are generated. The frequency with which each module occurs in the population can be tracked and the ones no longer in use by any population member may be removed from the library.

As previously stated, the average size of expressions in the population grows with time. This phenomenon is commonly referred to as *bloating*. Many theories have been proposed to deal with this size problem [1].
The defence against the crossover hypothesis suggests protecting critical sections of code and penalizing recombinations that are disruptive to important code sections.
The other hypothesis considers that bloating is a neutral phenomenon and occurs naturally in evolutionary processes.
We think that very complex programs do contain some inert code and some simplifications must be performed in order to avoid wasting cpu and memory consumption.
Introducing simplification rules means building a compiler which maps all semantically equivalent programs into a single object code.
The obvious alternative is to classify subexpressions according to their response to a set of test cases.

Bibliography

[1] Tackett, W. A. 1994. Recombination, Selection, and the Genetic Construction of Computer Programs. Doctoral Dissertation, University of Southern California, Faculty of the Graduate School, April 1994.

ANNEXE 1 : Declaration of classes Population, GP and Gene

```
class Population
{
        GP *pgpHeader;

        // total fitness of population
        FITNESS uliFitness;

        // total length of population
        unsigned long uliLength;

        FITNESS TotalFitness();
        unsigned long TotalLength();

        // Evaluation of the population
        void Evaluate();

        // Select randomly a member of the population
        GP* Select();

        // Tournament Select: returns best gp from tournament of size specified.
        GP* SelectBest();

        // Perform Selection and Crossover
        void Generate1();

        // Perform Mutation
        void Mutate();
};
```

```
class GP
{
        // Pointer to a pointer of genes which are the headers for each tree of the adfs
        Gene **ppgHeader;

        // variables used to store the fitness and length of the genetic program
        FITNESS iFitness;
        unsigned int iLength;

        ... function members ...

        GP();
        ~GP();

        // overwriting copy
        void Copy( GP* );

        // calculated the depth of a GP
        int Depth();

        // calculate length of GP both returning a value and setting uiLength
        int Length();

        // creates a genetic program with specific depth and method of creation
        void Create( unsigned int, int );

        // crosses two GPs altering the implicit object
        void Cross( GP*, GP* , int = 17 );

        // Mutates a GP
        void Mutate();

        // compares two GPs together
        unsigned int Compare( GP* );

        // evaluates a genetic program
        void Evaluate( int, int );

        // packs a genetic program as an S-expression
        void Pack(char*, Gene*);

        // builds the memory representation starting from an intermediate form
        void LoadStr(char *);

};
```

```
struct Gene
{
        // PGPS deals only with numerical identifiers
        unsigned int iValue;

        // numerical terminals and random numbers
        FITNESS forand;

        // pointers to child gene (if a function) and next gene (if part of a function arguments)
        Gene *pgChild,
               *pgNext;

        ... main function members ...

        // standard constructor
        Gene( unsigned int = 0, FITNESS = 0 );

        // overwriting copy
        void Copy( Gene* );

        // destructor
        ~Gene();

        // length of the subtree starting from this point
        void Length( unsigned int& );

        // depth of the subtree starting from this point
        unsigned int NewDepth(unsigned int);

        // returns the gene N starting from this gene
        Gene* Nth( unsigned int& );

        // chooses randomly a gene on a genetic program
        Gene* Choose();
}
```

ANNEXE 2 : Packing routines

```
void TranslateStr(char *buffer, Gene *pg)
{
	char itochar[10];

	// Numeric terminals

	if (pg->forand) {
		sprintf(itochar, FORMAT , (pg->forand) );
		strcat(buffer, itochar );
		return;
	}

	// Alphabetic terminals

	if (!pg->pgChild) {
		sprintf(itochar, "%s ", (pg->Name) );
		strcat(buffer, itochar );
		return;
	}

	// Function nodes

	sprintf(itochar, "( %s ", pg->Name );
	strcat(buffer, itochar);
}


void GP::Pack(char *buffer, Gene *pg)
{
	 TranslateStr(buffer, pg);
	 if (pg->pgChild) Pack(buffer, pg->pgChild);
	 if (pg->pgNext) Pack(buffer, pg->pgNext);
	 else strcat(buffer, ") ");
}
```

```
// Character string large enough to store a GP
char *PackedTree

void GP::Evaluate( int proc_id, int msg_id )
{
        Gene *root = *(this->ppgHeader);

        // Check if GP is just a .....Terminal node

        if ( ( !(root->pgChild) ) && ( !(root->pgNext) ) ) {
                strcpy(PackedTree, "( ( ");
                TranslateStr(PackedTree, root);
                strcat(PackedTree, ") ) ");
        }
        else {
                strcpy(PackedTree, "( ");
                this->Pack(PackedTree, *(this->ppgHeader) );
        }

        // Once packed, the GP identified by msg_id can simply be sent to node
        // identified by proc_id for fitness evaluation

        pvm_initsend(0);
        pvm_pkstr(PackedTree);
        pvm_send(proc_id, msg_id);
}
```

ANNEXE 3 : Unpacking and Evaluating a genetic program

```
int Gene::LoadStr( char* buffer, int left )
{
        char ch;
        char Atoi[10];
        int index = left;

        // input value of gene
        while (buffer[index] != ' ') {
                Atoi[index-left] = buffer[index];
                index++;
        }

        Atoi[index-left] = '\0';

        if   ( isnumber(Atoi) ) {
                forand = ATONUM(Atoi);
                iValue = forand;
        }
        else strcpy(Name,Atoi);

        index++;
        ch = buffer[index];

        // if we read in a 'c' = child create block and loop around for new child
        if ( ch == 'c' ) {
                index +=2;
                if ( !(pgChild = new Gene) ) ExitSystem( "Gene::LoadStr" );
                index = pgChild->LoadStr( buffer, index );
        }
        index += 2;
        ch = buffer[index];

        // if we read in a 'n' = next create block and loop around for new next
        if ( ch == 'n' ) {
                index +=2;
                if ( !(pgNext = new Gene) ) ExitSystem( "Gene::LoadStr" );
                index = pgNext->LoadStr( buffer, index );
        }

        // return the last used index.
        return(index);
}
```

The function Gene::*LoadStr*(char *, int) is called by the routine GP::*LoadStr*(char *).

```
void GP::LoadStr( char *buffer )
{
        // set up start of new Gene...
        Gene **ppg = ppgHeader;

        // loop through adfs allocating new genes and loading genetic trees

        for ( int i = 0; i < RootandADF; i++, ppg++ )
        {
            if ( !(*ppg = new Gene) ) ExitSystem( "GP::LoadStr" );
            (*ppg)->LoadStr( buffer, 0 );
        }
}
```

The function *LoadEval*(char *) builds and evaluates a genetic program by calling respectively the routines GP::*LoadStr*(char *) and *EvaluateFitness*(GP *).

```
FITNESS LoadEval(char *buffer)
{
        FITNESS Fitness;
         GP *pg;

         if ( !(pg = new GP) ) ExitSystem( "LoadEval()" );
         pg->LoadStr(buffer);

         Fitness = EvaluateFitness(pg);
         delete pg;
         return (Fitness);
}
```

The code below shows a complete example of a slave process illustrating how to receive, parse, build the memory representation and evaluate a genetic program.

```
#include "optimize.hpp"
int main()
{
        int mynum, proc_id, msg_type, buf_id, len;
        char str_tree[TREELENGTH], ObjCode[TREELENGTH];
        FITNESS Fitness;

        if ( (mynum = pvm_mytid() ) < 0 ) {
                cout<<"Failure enrolling the slave process.";
                pvm_exit();
                exit(-1);
        }
        else cout<<"The slave process "<<mynum
        <<" is ready to start."<<endl;

        InitialiseProblem();

        while(1) {
                buf_id = pvm_recv(-1, MSG_ALL);
                pvm_bufinfo(buf_id, &len, &msg_type, &proc_id);

                if (len >TREELENGTH ) {
                        cout << "Received too long message." << endl;
                        pvm_exit();
                        exit(0);
                };
                if (msg_type == MSG_KILL) {
                        pvm_exit();
                        exit(0);
                }
                if (msg_type >= 100)  {
                        pvm_upkstr(str_tree);
                        Parse(str_tree, ObjCode);
                        Fitness = LoadEval(ObjCode);

                        pvm_initsend(0);
                        pvm_pkbyte(  (char*)(&Fitness), sizeof(Fitness), 1 );
                        pvm_send(proc_id, msg_type);
                }
                else {
                        cout<<"Received erroneous message type "<<msg_type
                        <<" on process "<<mynum<<endl;
                        pvm_exit();
                        exit(-1);
                }
        }
}
```

ANNEXE 4 : Routines implementing Crossover operator

```
void GP::Cross( GP *mum, GP *dad, int maxdepthforcrossover )
{
        Gene *cutchild, *cutdad;
        unsigned int MaxDepth = 0;

        do {
        // copy the gp of mother into genetic tree of child
        Copy( mum );

        // select cut position of child from this new copied tree
        cutchild = (*(ppgHeader))->Choose();

        // select cut point from the same tree on the father...
        cutdad = (*( dad->ppgHeader ))->Choose();

        // Basically these next components are a splice operation for the GP
        // We copy into cutpoint of new child so this must be deleted if it exists
        if ( cutchild->pgChild ) delete cutchild->pgChild;

        // if dad had any children....
        if ( cutdad->pgChild ) {
                // copy this into new child
                if ( !(cutchild->pgChild = new Gene(cutdad->pgChild)) )
                ExitSystem( "GP::Cross" );
        }
        // otherwise set child to NULL
        else cutchild->pgChild = NULL;

        // copy dads values into mother....
        cutchild->iValue = cutdad->iValue;
        cutchild->forand = cutdad->forand;
        // Note we do not change pgNext value as this must stay the same...

        // Here we calculate the maximum depth of this new tree and only that tree.
        MaxDepth = 0;
        MaxDepth = (*(ppgHeader))->NewDepth(0);

        //  make sure that max depth is not mad depth of crossover
        } while ( MaxDepth > maxdepthforcrossover );

        // calculate the total length of this new individual.
        Length();
}
```

The function *NewDepth* returns the depth of a genetic program. It works as follows.

```
unsigned int Gene::NewDepth(unsigned int CurrentDepth)
{
	unsigned int left = CurrentDepth;
	unsigned int right = CurrentDepth;

	if ( pgChild ) left  = pgChild->NewDepth( CurrentDepth + 1 );
	if ( pgNext )  right = pgNext->NewDepth( CurrentDepth );
	return ( ( left > right ) ? left : right );
}
```

```
Gene* Gene::Choose()
{
	unsigned int iTotalLength = 0;
	Gene *pg = NULL;

	// calculate the total length of the whole genetic program so far starting at this gene
	Length( iTotalLength );

	// loop 10 times
	for ( int i = 0; i < 10; i++ ) {
		// calculate a random number between 1-> TotalLength
		unsigned int iLengthCount = ( gp_rand() % iTotalLength ) + 1;

		// find gene with this value.....
		pg = Nth( iLengthCount );

		// if this pointer points to a function then return this value
		// else keep going around the loop
		if ( pg->pgChild ) return pg;
	}

	// if after 10 loops still don't have function o/p terminal
	return pg;
}
```

The function *Length* calculates the length (complexity) of a genetic program.

```
void Gene::Length( unsigned int& riLength )
{
        // increment length reference variable
        riLength++;

        // check for child and next member and loop back through function if they occur....
        if ( pgChild ) pgChild->Length( riLength );
        if ( pgNext )  pgNext->Length( riLength );
}
```

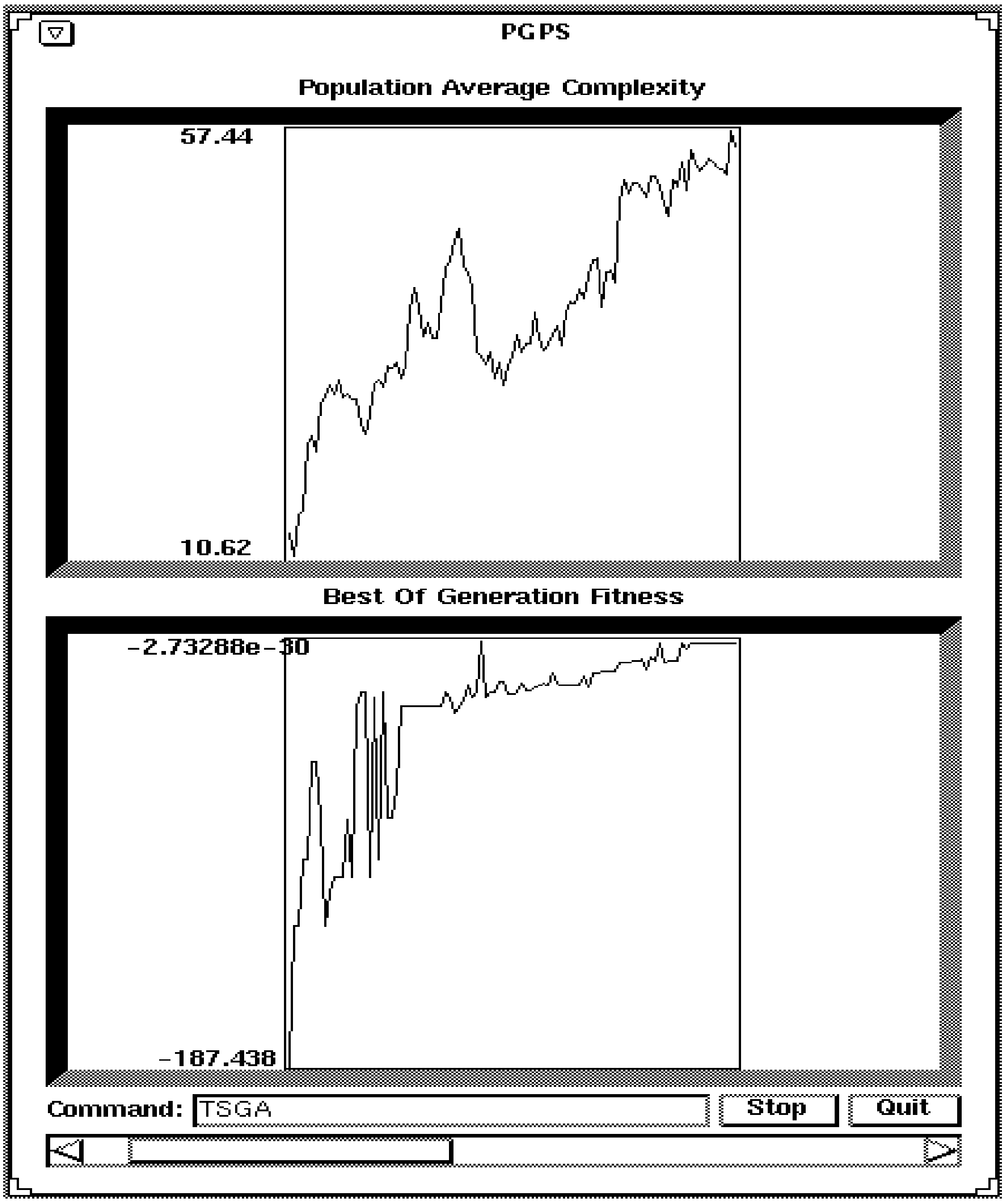
The function *Nth* returns the *Nth* gene of the GP.

```
Gene* Gene::Nth( unsigned int &iLengthCount )
{
        // decrement the length so far and return if this is == 0
        if ( --iLengthCount == 0 ) return this;

        // set up pointer to a gene to check if we find a solution
        Gene *pg = NULL;

        // if this gene has a child loop back around to function
        if ( pgChild ) pg = pgChild->Nth( iLengthCount );

        // check if this gene has a next and also that the solution was not found in
        // the previous child section and try to find it in next member
        if ( ( pgNext ) && ( pg == NULL ) ) pg = pgNext->Nth( iLengthCount );

        // return pointer to gene which will contain correct pointer at the end
        return pg;
}
```

ANNEXE 5 : Evaluating a genetic program

```
FITNESS Translate( Gene *pg )
{
        FITNESS tmp = 0;

        // Numeric terminal nodes
        if (tmp = pg->forand) return (tmp);

        // Alphabetic terminal nodes
        if (!pg->pgChild)
                if ( !strcmp(pg->Name, "X") ) return(tmp = globalX);

        // Function nodes
        if ( !strcmp(pg->Name, "*") ) {
                tmp= Translate( pg->pgChild ) * Translate( pg->pgChild->pgNext );
                return (tmp);
        }

        if ( !strcmp(pg->Name, "+") ) {
                tmp= Translate( pg->pgChild ) + Translate( pg->pgChild->pgNext );
                return (tmp);
        }

        if ( !strcmp(pg->Name, "-") ) {
                tmp= Translate( pg->pgChild ) - Translate( pg->pgChild->pgNext );
                return (tmp);
        }

        if ( !strcmp(pg->Name, "/") ) {
                tmp= DIV( pg->pgChild );
                return (tmp);
        }

}

// This is a protected division
FITNESS DIV( Gene *pg)
{
        FITNESS numer = Translate( pg );
        FITNESS denom = Translate( pg->pgNext );

        if ( denom == 0 ) return 1;
        else return (numer / denom);
}
```

PGPS graphical tool



The graphical tool (gt) for PGPS is developed using Tcl (Tool command language) and Tk (X windows toolkit). The gt interface plots two curves in evolution while PGPS runs. The upper curve represents, by generation, the population average complexity. The lower curve shows the fitness of the best-of-generation genetic program.

Bibliography

[1] Agarwal, A. 1992. Performance Tradeoffs in Multithreaded Processors, IEEE Transactions on parallel and distributed systems, vol. 3, no. 5, September 1992.

[2] Allen, F. and Karjalainen R. 1993. Using genetic algorithms to find technical trading rules. Technical report, Wharton school, university of Pennsylvania.

[3] Baker, J. 1985. Adaptative Selection Methods for Genetic Algorithms. *Proc. 1st ICGA*, June 1985.

[4] Bennett, F. H. 1996. Automatic creation of an Efficient Multi-Agent Architecture Using genetic programming with Architecture-Altering Operations. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996,* pages 30-38. MIT Press, 1996. Proceedings of the first Annual Conference, July 28-31, 1996, Stanford University.

[5] Bergmann, S. 1994. Compiler Design: Theory, Tools and Examples, WCB Publishers.

[6] Davis, L. 1991. Handbook of Genetic Algorithms. Van Nostrand Reinhold, 1991.

[7] De Jong, K. 1975. The Analysis and behaviour of a Class of Genetic Adaptive Systems. PhD thesis, University of Michigan, 1975.

[8] Fraser, A. P. 1994. Genetic programming in C++, public domain genetic programming system.

[9] Geist, A., Beguelin A., Dongarra J., Jiang W., Manchek R. and Sunderam V. 1994. PVM3 User’s Guide and reference manual, ORNL/TM-12187, September 1994.

[10] Goldberg, D.E. 1989. Genetic Algorithms in search, optimization and machine learning. Addison-Wesley, 1989.

[11] Hilborn, R. C., 1994. Chaos and Nonlinear dynamics, Oxford University Press.

[12] Holland, J. 1975. Adaptation in Natural and Artificial Systems, University of Michigan Press, Ann Arbor.

[13] Holland, J.H. 1992. Adaptation in Natural and Artificial Systems. MIT Press, 1992.

[14] IBM, 1994. IBM AIX PVMe User’s guide and Subroutine Reference, December 1994.

[15] IBM, 1995. IBM *System Journal*, 34(2), 1995.

[16] Kirkpatrick, S., Gelatt C.D. and Vecchi M.P. 1983. Optimization by Simulated Annealing. Science, 220, 671 (1983).

[17] Koza, J. 1992. Genetic programming, MIT Press.

[18] Koza, J. 1994. Genetic programming II, Automatic Discovery of Reusable Programs, MIT Press.

[19] Koza, J. and Andre D. 1995. Parallel Genetic Programming on a Network of Transputers, Computer Science Department, Stanford University, Technical report CS-TR-95-1542.

[20] Manderick, B. and Spiessens P. 1989. Fine Grained Parallel Genetic Algorithms. *Proc. 3rd ICGA*, June 1989.

[21] Michalewicz, Z. 1992. Genetic Algorithms + Data Structures = Evolution Programs. Springer-Verlag, 1992.

[22] Muhlenbein, H. 1991. Evolution in Time and Space - The Parallel Genetic Algorithm. Foundations of genetic algorithms, G. Rawlins, ed. Morgan-Kaufmann.

[23] Oussaidène, M. and Chopard B. 1994. Optimisation des expressions arithmétiques sur une machine massivement parallèle en utilisant un algorithme génétique, SIPAR-Workshop on “Parallel and Distributed Computing”, university of Fribourg, October 1994.

[24] Oussaidène, M., Chopard B. and Tomassini M. 1995. Programmation évolutionniste parallèle, RenPar’7 , PIP-FPMs Mons , Belgium. Libert G., Dekeyser J.L., Manneback P. (editors).

[25] Oussaidene, M., Chopard B., Pictet O. V. and Tomassini M. 1996. Parallel genetic programming: an application to trading models evolution. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996*, pages 357-362. MIT Press, 1996. Proceedings of the first Annual Conference, July 28-31, 1996, Stanford University.

[26] Oussaidene, M., Chopard B., Pictet O. V. and Tomassini M. 1996. Parallel Genetic Programming and its application to trading model induction. Submitted to the journal *Parallel Computing*, July 1996 .

[27] Pictet, O.V. , Dacorogna M.M. , Muller U. A. , Olsen R. B. and Ward J.R. 1992. Real-time trading models for foreign exchange rates , Neural Network World 6/92, 713-744.

[28] Pictet, O.V., Dacorogna M.M., Chopard B., Oussaidene M., Schirru R. and Tomassini M. 1995. Using Genetic Algorithms for Robust Optimization in Financial Applications , Neural Network World 4/95, 573-587.

[29] Pring, M. J. 1988. Technical Analysis Explained, McGraw-Hill.

[30] Punch, W.F., Goodman E.D., Pei M., Chia-Shun L., Hovland P. and Enbody R. 1993. Further Research on Feature Selection and Classification Using Genetic Algorithms, Appeared in ICGA93, pg 557-564, Champaign III.

[31] Ross, S. J., Daida J. M., Doan C. M., Bersano-Begey T. F. and McClain J. J. 1996. Variations in Evolution of Subsumption Architectures Using Genetic Programming: The Wall Following Robot Revisited. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996*, pages 191-199. MIT Press, 1996. Proceedings of the first Annual Conference, July 28-31, 1996, Stanford University.

[32] Sitllger, M. and Spiliopoulou M. 1996. Genetic programming in Database Query Optimization. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996*, pages 388-393. MIT Press, 1996. Proceedings of the first Annual Conference, July 28-31, 1996, Stanford University.

[33] Syswerda, G. 1989. Uniform Crossover in Genetic Algorithms. *Proc. 3rd ICGA*, June 1989.

[34] Tackett, W. A. 1994. Recombination, Selection, and the Genetic Construction of Computer Programs. Doctoral Dissertation, University of Southern California, Faculty of the Graduate School, April 1994.

[35] Tanese, R. 1989. Distributed Genetic Algorithms. *Proc. 3rd ICGA,* June 1989.

[36] Welch, B. 1995. Practical Programming in Tcl and Tk. DRAFT, January 13, 1995.

[37] Whitley, D. 1989. The GENITOR Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials Is Best. *Proc. 3rd ICGA*, June 1989.

[38] Whitley, D. 1993. A Genetic Algorithm Tutorial. Technical Report CS-93-103, Colorado State University, November 1993.