

# Programmation Génétique

*Denis Robilliard*

(Laboratoire d'Informatique du Littoral,  
Univ. Littoral-Côte d'Opale, Calais)

Ecole JET 2007, Yravals, France

# Plan

- Bases de la PG
  - Définition
  - Historique
  - La PG « canonique » de Koza
  - Un exemple de mise en œuvre et de paramétrage
  - Applications
- Compléments
  - Typage, autres représentations, modularité
  - Accélération de la PG
  - Introns et congestion
  - Théorème des schémas, rôle du cross-over,

# Définition

- La Programmation Génétique (PG) :
  - **génération automatique de programmes (J. Koza)**
  - **génération automatique de comportements représentés par des programmes exécutables (P. Angeline).**
- Ces définitions ne précisent pas « par une approche évolutionnaire ». On peut plutôt parler de **programmation automatique** (J. Koza, W. Banzhaf, ...) .
- Cependant une majorité de travaux s'inspirent du paradigme des algos génétiques.

# Quelques jalons historiques

- 1958, Friedberg : mutation aléatoire d 'instructions , attribution de « crédits » aux instructions des programmes les plus efficaces.
- 1963, Samuel : utilisation du terme « machine learning » dans le sens de programmation automatique.
- 1966, Fogel, Owen, Walsh : automates à états finis pour des tâches de prédiction, obtenus par sélection de parents efficaces auxquels on applique des mutations : *Evolutionary programming*.

- 1985, Cramer : utilisation d'expressions sous forme d'arbres Cross-over entre sous-arbres.
- 1986, Hicklin : évolution de programmes de jeu en LISP. Sélections de parents efficaces, combinaisons des sous arbres communs ou présents dans un des parents et de sous-arbres aléatoires.
- 1989, 1992, Koza : Systématisation et démonstration de l'intérêt de cette approche pour de nombreux problèmes. Définition d'un paradigme standard dans le livre « Genetic Programming. On the Programming of Computers by Means of Natural Selection » [Koza, 1992].

# La PG instanciatiion des AGs

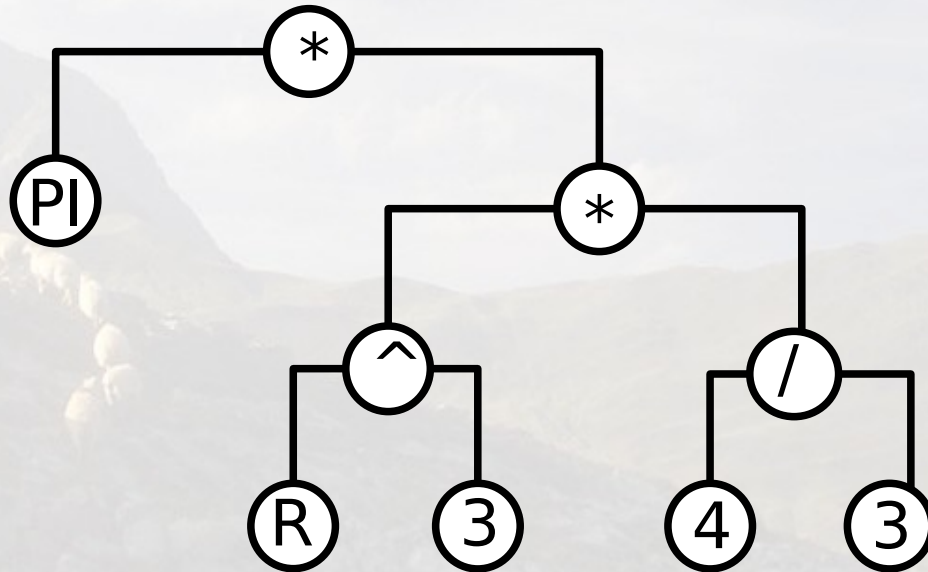
- PG = AG où 1 solution = 1 programme
- On retrouve les problématiques des AGs :
  - problème de représentation des solutions
  - choix de la mesure de qualité (fitness)
  - réglage de la pression de sélection
  - taille de la population, nombre de générations
  - opérateurs génétiques
  - coût calculatoire
  - ...

# La PG « canonique » (Koza)

- *Dans ce modèle, la PG est une instanciation du paradigme des algos génétiques*
  - programmes structurés en expressions arborescentes.
  - définition d'un ensemble de fonctions primitives et de terminaux, constituant les expressions.
  - type de retour unique pour toutes les expressions.
  - évolution via les mécanismes de l'échange (crossover) de sous-arbres, pondération pour minimiser l'échange de feuilles au profit de l'échange de sous-arbres plus grands.
  - rôle très réduit, voire inexistant, des mutations aléatoires.
  - limitation de la profondeur des arbres due aux contraintes d'implémentation

# Représentation

- Expressions « LISP » :  
( \* PI ( \* ( ^ R 3 ) ( / 4 3 ) ) )
- arbres équivalents :



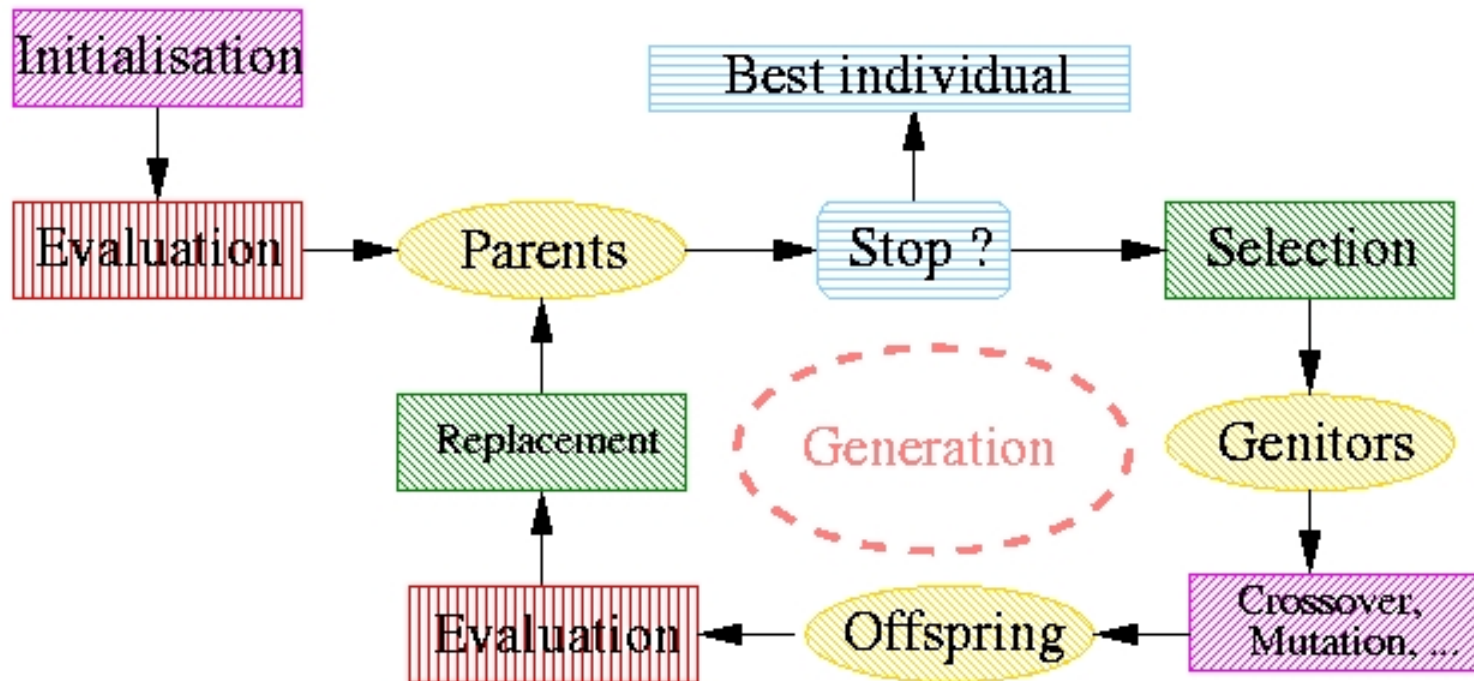
- Ces arbres peuvent être implémentés dans d'autres langages (C, C++, Java, ...)




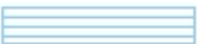
# Terminologie

- Les terminaux (feuilles de l'arbre) :
  - pseudo-variables contenant les entrées du programme
  - constantes, fixées d 'après la connaissance préliminaire du problème, ou générées aléatoirement :ERC (*ephemeral random constants*)
  - fonctions sans arguments mais avec effets de bord
  - variables ordinaires (*note : l'approche fonctionnelle à la LISP se dispense souvent de la présence de variables*)

- Les fonctions ou opérateurs (nœuds internes de l'arbre) :
  - exemple : fonctions booléennes, arithmétiques, transcendentales, à effet de bord (assignation de variables, déplacement d'un robot, ...), fonctions implantant des structures de contrôle : alternative, boucle, appel de routines, ...
  - préférer un ensemble de fonctions petit et bien ajusté au domaine du problème, pour réduire l'espace de recherche. Attention à ne pas le réduire trop, sous peine de perdre la possibilité de trouver des solutions intéressantes !

# Schéma général AG/PG



-  **Stochastic operators: Representation dependent**
-  **"Darwinism" (stochastic or determinist)**
-  **Main CPU cost**
-  **Checkpointing: stopping criterion and statistics**

# Population initiale

- On fixe une profondeur maximale pour les arbres.
- Création d'arbres aléatoires par 2 méthodes principales :
  - « grow » : chaque nœud est tiré dans l'ensemble {terminaux} + {fonctions}
  - les arbres sont de forme irrégulière
  - « full » : on ne peut tirer un terminal que lorsque l'on est à la profondeur maximum
  - arbres équilibrés et « pleins »

- Une synthèse, la méthode « ramped half & half » :
  - on va générer équitablement des arbres avec des profondeurs régulièrement échelonnées :  
2, 3, 4, ..., maximum
  - à chaque profondeur, une moitié est générée par la méthode « full », l'autre par la méthode « grow »
- L'objectif est d'obtenir plus de variabilité dans la population. C'est la méthode préférentielle actuellement (*cependant voir travaux de R. Poli*).

# Evaluation du “fitness”

- Comment évaluer la performance d'un programme ? Tout dépend du problème.
- Quelques exemples :
  - contrôle d'un robot : nombre de chocs contre les murs
  - classification : nombre d'exemples bien classés
  - vie artificielle : quantité moyenne de nourriture ingérée dans une simulation
  - régression : mesure d'erreur sur un jeu d'exemples
- En général, plusieurs mesures de qualité sont possibles
- Chaque exemple est encore appelé « *fitness case* »

- Exemples de fonctions fitness usuelles pour les problèmes de régression :

- somme des valeurs absolues des écarts entre valeur calculée par le programme et valeur attendue en sortie, pour chacun des *fitness cases* :

$$\sum_{i=1}^n |P(e_i) - s_i|$$

- somme des carrés des écarts entre valeur calculée et valeur attendue (squared error) :

$$\sum_{i=1}^n (P(e_i) - s_i)^2$$

*« Une fonction fitness idéale devrait renvoyer une mesure différenciée et continue de l'amélioration de la qualité des programmes » (Banzhaf et al.).*

Un peu de terminologie :

- « fitness standardisé » (fs): la valeur du meilleur fitness possible est 0, toutes les valeurs de fitness sont positives.
- « fitness normalisé » : fitness standardisé où la valeur du fitness est toujours comprise entre 0 et 1.
- « fitness ajusté » (fa), un fitness normalisé où le meilleur score possible vaut 1. Souvent on prend  $1/(1+\text{fitness\_standardisé})$ .

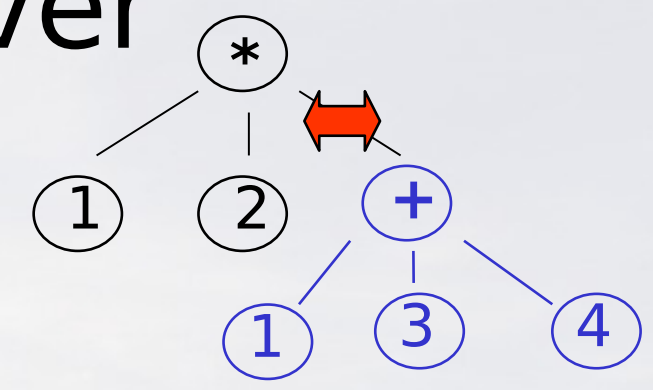
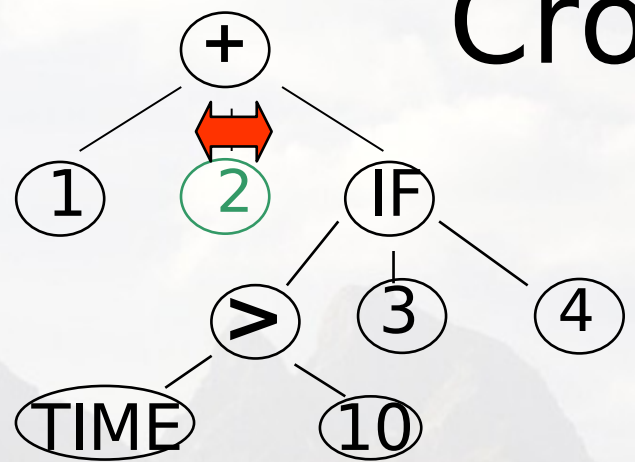
# La sélection

- On retrouve les méthodes de sélection utilisées dans les AGs :
  - sélection proportionnelle au fitness, avec éventuelle normalisation (scaling)
  - sélection basé sur le rang de l'individu dans la population (ranking)
  - sélection par tournoi : la plus courante, car rapide et facilement parallélisable

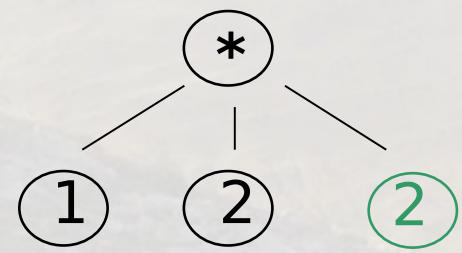
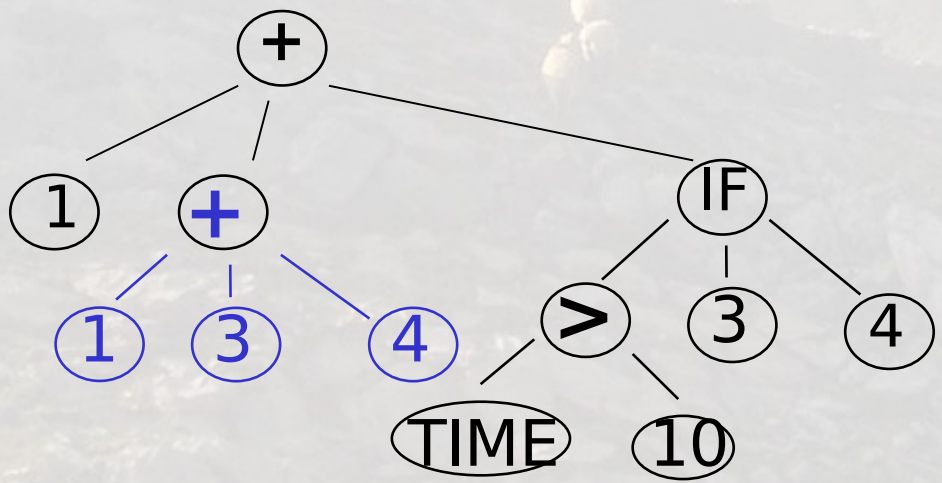
# Opérateurs Génétiques : Copie

- Simple recopie d'un individu d'une génération à la suivante.
- Peut être forcée pour le meilleur individu: élitisme.

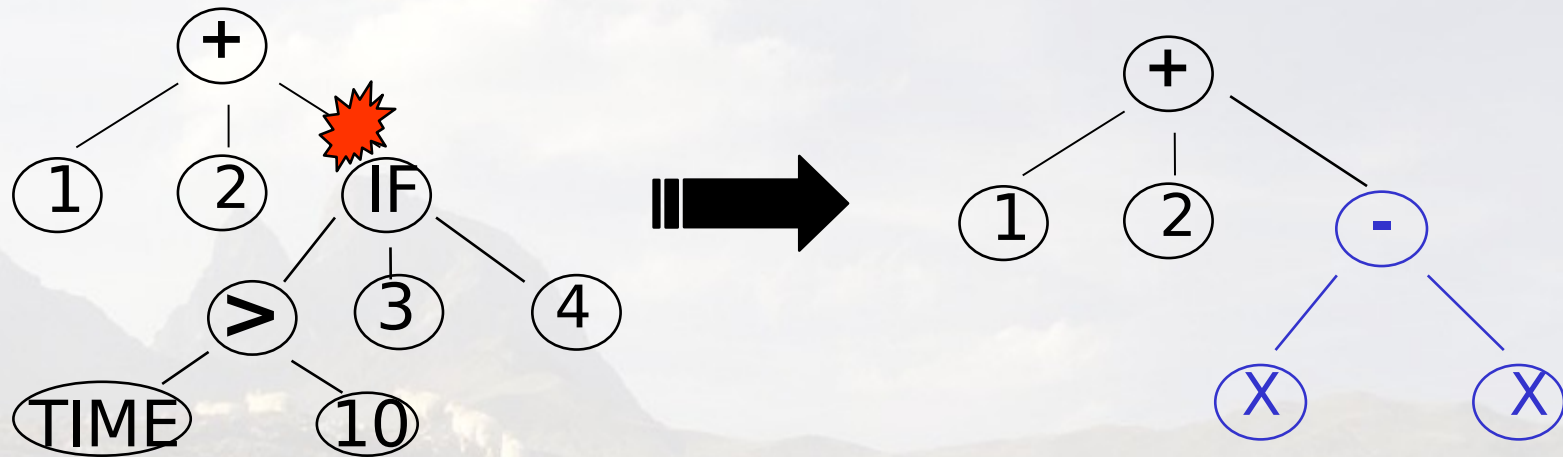
# Opérateurs Génétiques : Cross-Over



- Echange de deux sous-arbres pris aléatoirement



# Opérateurs Génétiques : Mutation



Destruction d'un sous-arbre et remplacement par un sous-arbre aléatoire, créé comme lors de la génération de la population initiale.

# Note sur les opérateurs génétiques

- Le cross-over ou la mutation sont susceptibles de transformer n'importe quel sous-arbre argument d'une fonction.
- Les fonctions doivent donc être capables d'accepter toutes sortes de valeurs en argument
- On impose souvent un type unique de valeurs retournées (*propriété de clôture*)
- *Exemple : remplacer la division standard par la "division protégée" qui renvoie 0 (ou un grand entier) en cas de division par 0.*

# Le Remplacement

- Façon « AG » :
  - Générationnel : la nouvelle génération est souvent de même taille que l'ancienne et elle la remplace.
  - « Steady state » : chaque nouvel individu est inséré dans la population lors de sa création et remplace un individu déjà présent, par exemple celui de plus mauvais fitness ou le moins bon du tournoi si sélection par tournoi.

# Un exemple de mise en œuvre

(© Banzhaf *et al.*)

- Soit un ensemble de cas de fitness représentant le problème (créé artificiellement à partir de  $f(x) = x^2 / 2$ )

<i>cas</i>	<i>x</i>	<i>y=f(x)</i>
1	0,1	0,005
2	0,2	0,020
3	0,3	0,045
4	0,4	0,080
5	0,5	0,125
6	0,6	0,180
7	0,7	0,245
8	0,8	0,320
9	0,8	0,320
10	1	0,500

# Préparation du run

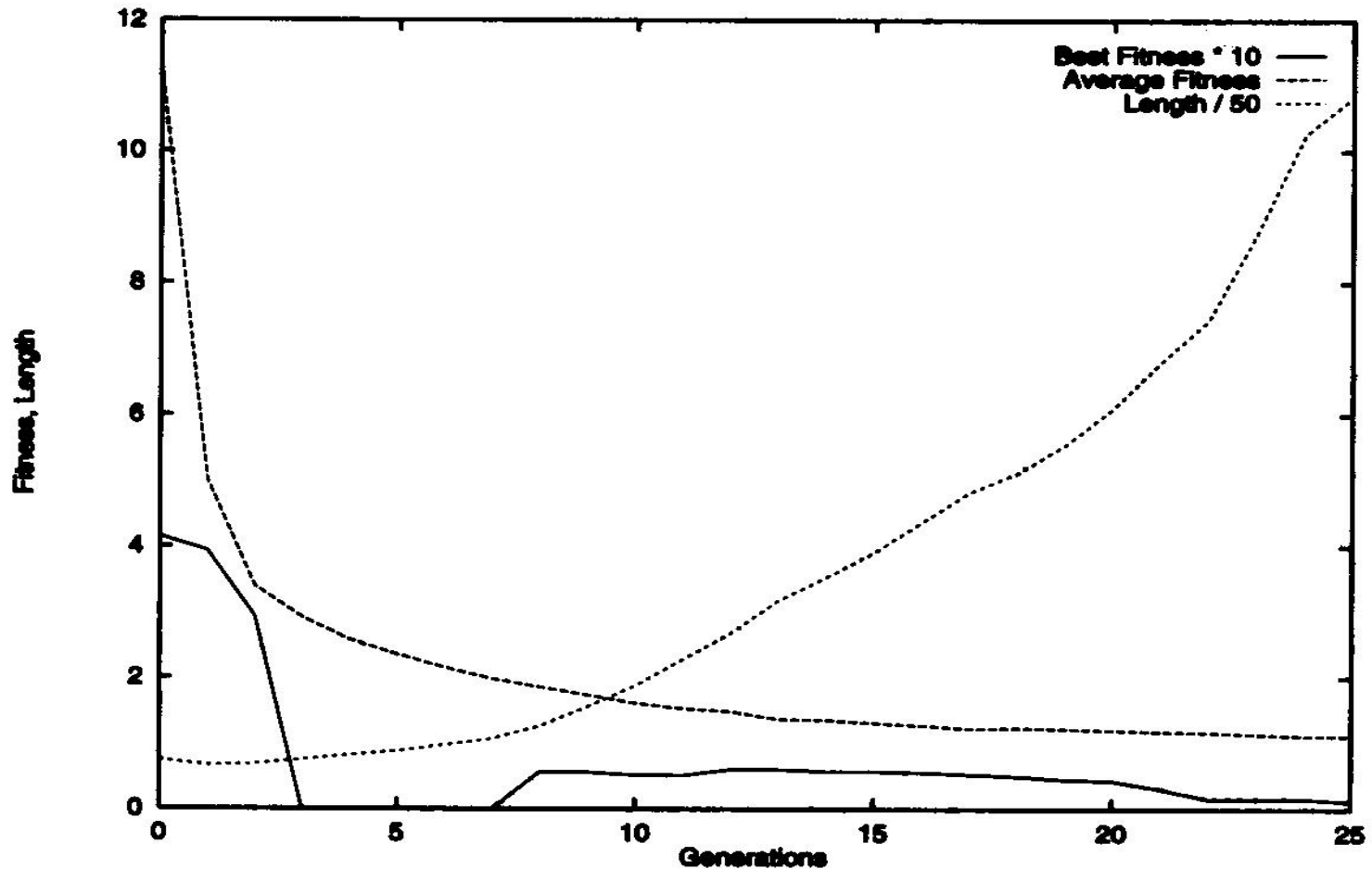
- Décider des paramètres :
  - Terminaux : X, constantes entières  $\leq 10$
  - Fonctions : +, -, \*, /
  - Taille de population = 100
  - Nombre maximum de générations = 25
  - Proba de cross-over 80%
  - Proba de mutation 10%
  - Pas d'élitisme
  - Initialisation par méthode Grow (prof max 4)
  - Profondeur maximum 10
  - Sélection par tournois de taille 5
- Paramètre implicite :
  - Générateur aléatoire

# Déroulement du run

- meilleur individu :
  - gén. 0 :  $X/3$
  - gén. 1 :  $X/(6-3X)$
  - gén. 2 :  $X/(X(X-4)-1+4/x-((9(X+1))/5X - \dots$
  - Gén. 3, 4 & 5 :  $X*X/2$  (optimum trouvé)
  - Gén. 6 à 25 : optimum perdu (pas d'élistisme)

# Déroulement du run

- Evolution du fitness et de la taille



# Paramétrage type

- Populations plutôt grosses : 500 jusqu 'à 1.000.000 arbres
- Nombre max de générations plutôt faible : 50 à 100
- Parsimonie dans le choix du langage (nombre de terminaux et de fonctions)
- Utiliser des fonctions permettant un comportement non linéaire
- Pas trop de pression de sélection, ex : tournoi de taille 4
- Peu de mutation  $< 5 \%$

# Applications de la PG

- Tri (Kinneer), gestion de caches (Paterson et al.), compression de données (Nordin et al.), ...
- Reconnaissance d'images (Robinson et al.), classification d'images (Zao), traitement d'images satellitales (Daïda), ...
- Prédiction de séries temporelles (Lee), génération d'arbres de décisions (Koza), datamining (Freitas), ...
- Classification de segments d'ADN (Handley), de protéines (Koza et al.), ...
- Synthèse de circuits électroniques (Koza),
- Planification de déplacements de robot (Faglia et al.), évitement d'obstacles (Reynolds) , mouvement de bras robotisés (Howley), ...
- Modélisation par éléments finis (Schoenauer et al.)...

- John Koza s'est intéressé aux applications où la PG fait au moins aussi bien que les humains :
  - « Human competitive A.I. »
- Obtention de résultats brevetables:
  - synthèse de circuits électroniques (Koza)
  - programmation d'ordinateurs quantiques (Lee Spector)
  - session dédiée à GECCO...

# Compléments

The background of the slide is a faded, sepia-toned photograph of a mountainous landscape. In the foreground, a rustic stone wall runs across the frame. Beyond the wall, there are rolling hills and several sharp, rocky mountain peaks under a sky filled with soft, white clouds. The overall aesthetic is that of a historical or geographical document.

# Typage

- Introduction du typage fort (Montana 1995, Gruau 1996) : toutes les fonctions n'acceptent pas et ne retournent pas un même type de valeurs.
- Peut se gérer par « essai et erreur » (ex : librairie ECJ) :
  - répéter le choix aléatoire d'un point de cross-over jusqu'à ce que le type de retour soit correct.
- Par utilisation de grammaires explicites (voir aussi Whigham 97, Wong 95).

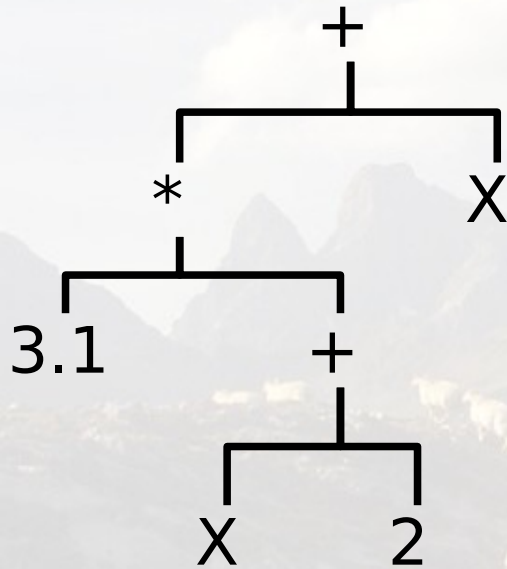
# Représentations avec grammaires explicites

- Grammaire algébrique (context-free) :
  - Non terminaux =  $\{exp, op, var\}$
  - Terminaux =  $\{+, *, X, ERC, (, )\}$
  - Règles de productions =
    - $start ::= exp$
    - $exp ::= ( op exp exp ) | val$
    - $op ::= + | *$
    - $val ::= X | ERC \}$
  - Utilisation typique : syntaxe des langages de programmation.

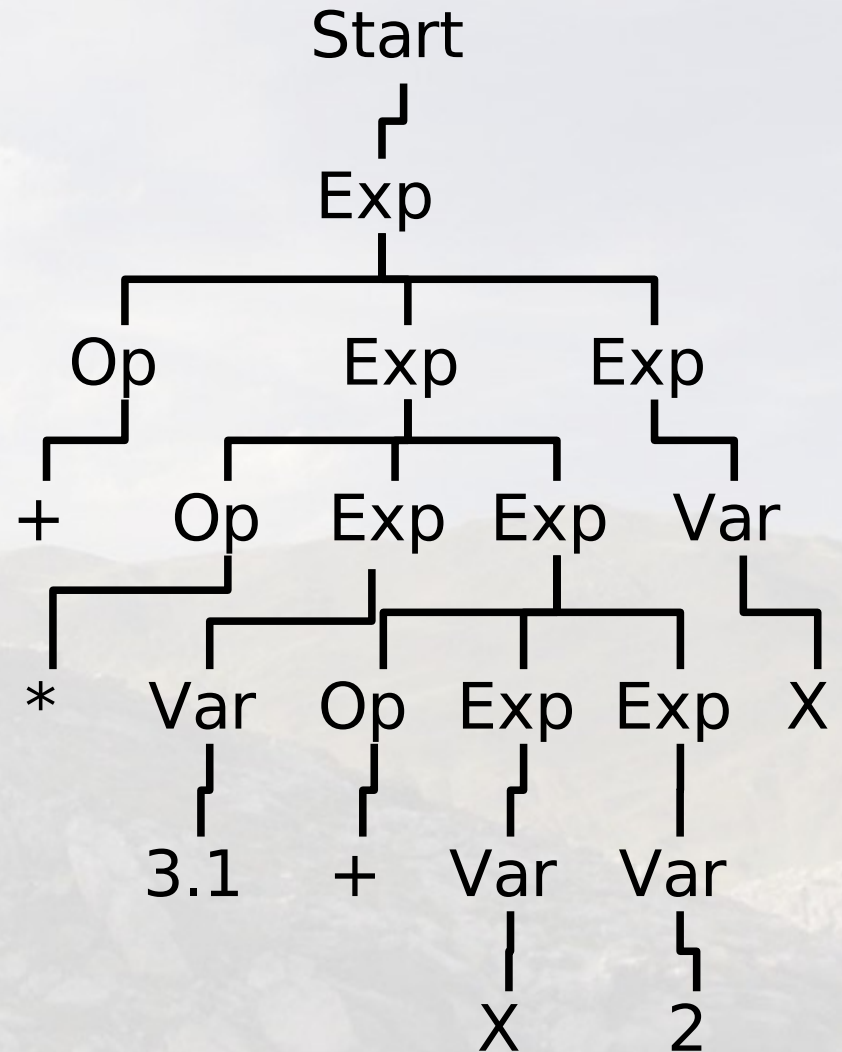
# Grammaires closes / non closes

- GP classique = grammaire close implicite, peut s'exprimer en une seule règle :  
$$start = ERC \mid X \mid (+ \ start \ start) \mid (* \ start \ start)$$
- Grammaires non closes :  
$$start = ERC \mid X \mid (+ \ start \ start) \mid (* \ start \ start) \mid (if \ bool \ start \ start)$$
  
$$bool = true \mid false \mid (<= \ start \ start) \mid (! \ bool) \mid (\&\& \ bool \ bool) \mid (\|\| \ bool \ bool)$$
- Les grammaires non closes peuvent servir à introduire le typage

# Arbres de dérivation



*arbre/solution  
usuel*



*arbre de dérivation / grammaire*

- Représentation :
  - ex : numéroter les productions associées à une règle, génôme = chaîne d'entiers indiquant successivement quelles productions sont dérivées (Grammatical Evolution (GE) Ryan & O'Neill 98).
- Opérateurs génétiques qui respectent les contraintes (ou pas ex: GE !) :
  - ex : une "expression" est forcément remplacée par une autre "expression"
  - => conserver les types.
  - => autoriser le mixage et/ou la conversion de certains types (Ratle & Seebag 2001), ex : couple = newton\*mètre.

# Représentation linéaire

- Utiliser un langage machine avec une syntaxe régulière, les données étant chargées dans un jeu de registres (Banzhaf et al. 97 (?)).  
Load B, 1  
Load C, 2  
Load A, TIME  
Gtr 10, +3  
Load A, 4
- Le résultat est rangé dans un registre spécifique (ici A) ou envoyé sur un périphérique de sortie.  
Jmp +2  
Load A, 3  
Add A, B  
Add A, C
- Cross-over des AGs pour les structures linéaires :  
1-point, 2-points, ...

# Représentation linéaire

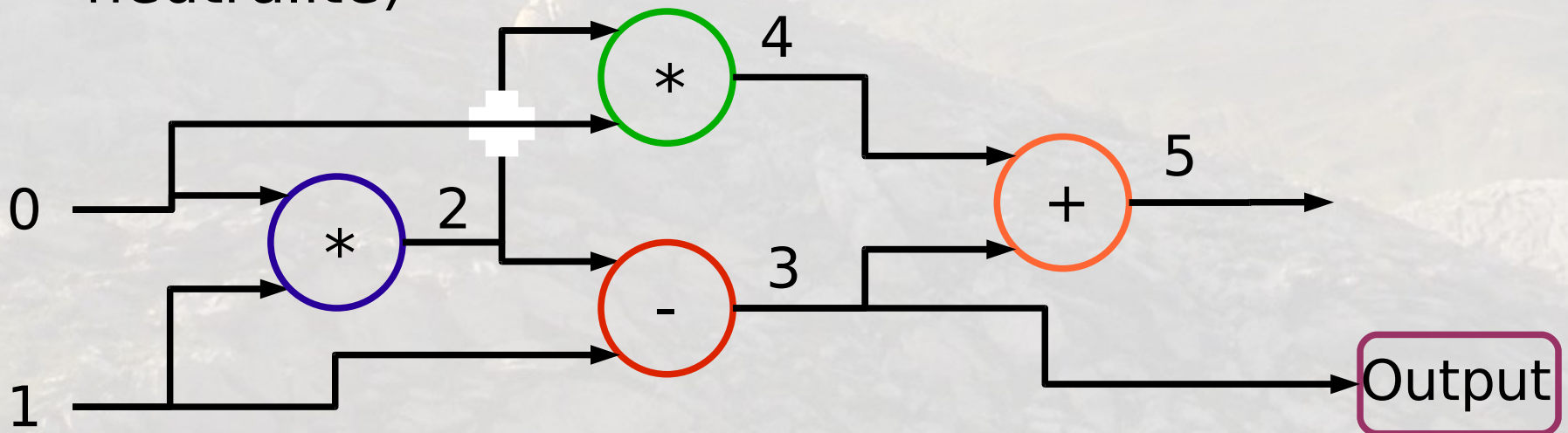
- Le principe peut être étendu à un sous-ensemble du langage C.
- Assembleur linéaire : utilisé dans le progiciel Discipulus (Banzhaf, Nordin) pour la régression et la classification.
- Utilisation de machine à pile (Perkins 94)
  - ex : PushGP (Lee Spector) ou bytecode Java
  - Quelle valeur utiliser de celles qui restent sur la pile ?

# Cartesian GP

- Julian Miller & Thomson 2000, autre forme de représentation linéaire (simplifiée en 2001) :
  - individu = chaîne d'entiers de taille fixe.
  - l'individu s'interprète comme une séquence de n-uplets (n fixé = 1 + arité max des op) terminée par un entier unique.
  - 1 n-uplet = { # argument1, # argument2, ..., # argument (n-1), # fonction }
  - on numérote de 0 à (k-1) les k entrées du programmes.
  - les sorties de chaque noeud sont ensuite numérotées à partir de k.
- L'évolution se fait par mutation, sur le principe des Stratégies Evolutionnaires (Schwefel et al.)

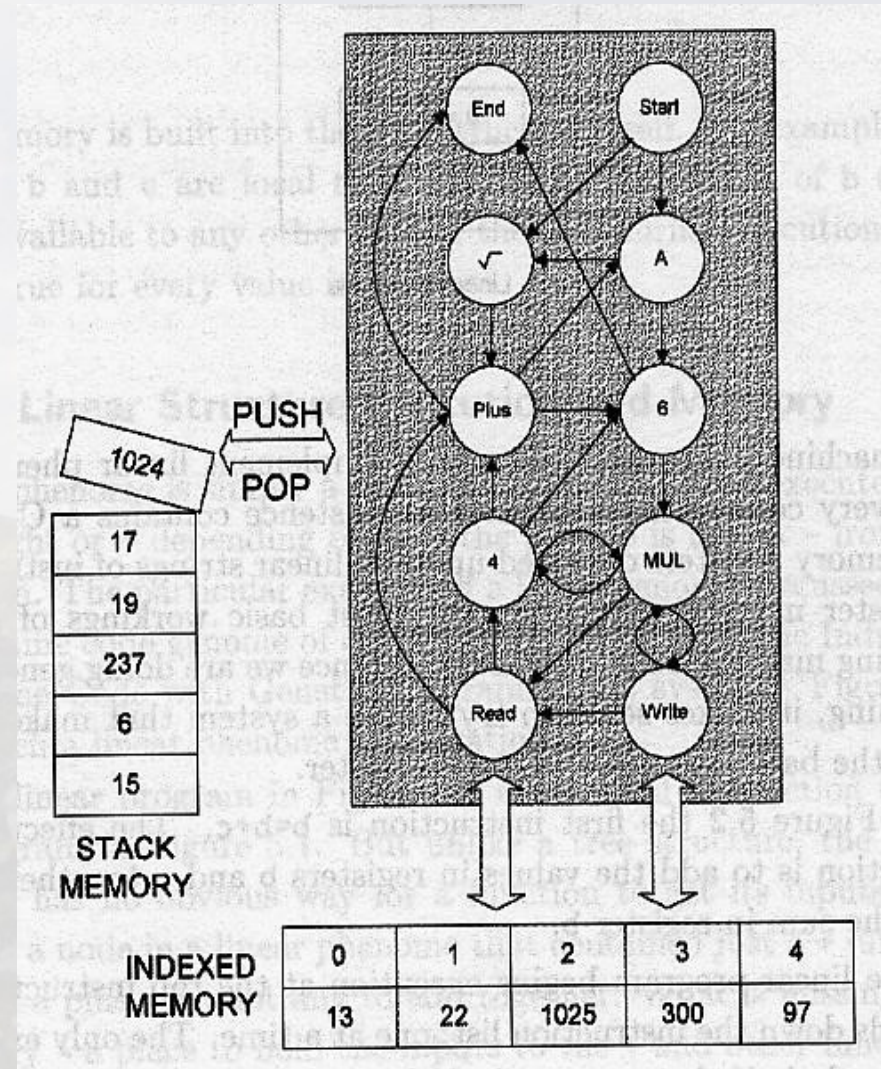
# Cartesian GP : exemple

- 2 entrées : # 0, 1
- fonctions/# : +/0, -/1, \*/2
- Exemple de programme : {0,1,2} {2, 1, 1} {2,0,2} {4, 3, 0} 3
- On génère le graphe dirigé acyclique correspondant; notez que certaines sorties peuvent être inutilisées (présence de neutralité)



# Représentation en graphe

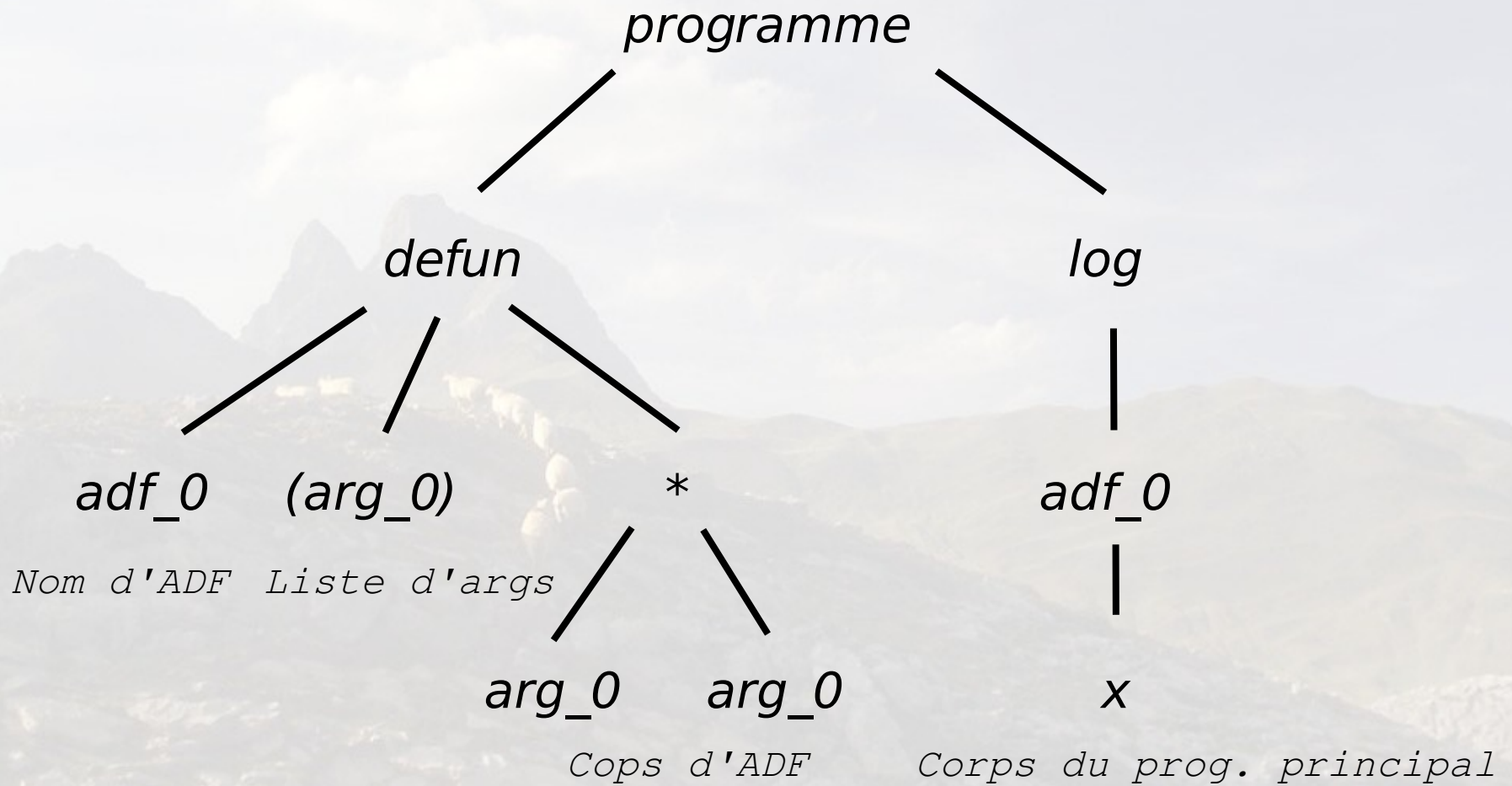
- Le système PADO (Teller & Veloso, 1995)
- On calcule avec les valeurs dans la pile.
- Des instructions permettent de lire et d' écrire entre la pile vers la mémoire.
- Un jeu de règles détermine le prochain nœud du graphe en fonction de valeurs prises dans la pile ou la mémoire.
- Opérateurs génétiques spécifiques



# La modularité

- Modularité et ré-utilisabilité sont des notions essentielles en génie logiciel.
- Koza (1992, 1994) : notion d'ADF (Automatically Defined Functions).
- Angeline, Pollack (1992, 1993) : extraction automatique de sous-programmes, conservés dans une bibliothèque.
- Sous-programmes implicites dans certaines approches (code machine)

# Les ADFs



# Les ADFs (suite)

- Le langage (terminaux plus fonctions) et les paramètres génétiques (taux de cross-over, ...) sont fixés indépendamment pour le programme et pour chaque ADF.
- Ces choix, en particulier celui du langage de l'ADF, peuvent être intéressants pour attaquer un problème que l'on sait composite.

# Extraction de routines

- Un sous-arbre est sélectionné, jusqu'à une certaine profondeur, et on lui attribue un nom de fonction. La partie en dessous est considérée comme les arguments de la fonction. Il est recopié dans une bibliothèque et remplacé par un appel fonctionnel.
- Le mécanisme inverse permet de réintroduire le code dans l'arbre et donc de le soumettre à nouveau à évolution.
- Lorsqu'une telle fonction n'est plus employée, elle est retirée de la bibliothèque.

# Introns et congestion

- Intron = morceau de génotype n'apportant pas de contribution lors de l'évaluation du programme.
  - Intron syntaxique = n'est pas évalué. Exemple : branchement conditionnel toujours faux
    - Exemple : (if (false) (code))
  - Intron sémantique = est évalué (et coûte du temps machine...) mais ne change rien au fitness.
    - Exemple :  $x * (3 - 2 * 1)$

- La congestion (*bloat*, *bloating*) désigne spécifiquement la taille exagérée d'une expression par rapport aux possibilités du langage utilisé.
  - Exemple :  $1 + 1 + 1 + 1$  est plus congestionné que  $2 + 2$  si le langage permet les deux expressions.
- Les introns induisent la congestion mais n'en sont donc pas la seule raison (en effet l'exemple précédent peut influencer sur le fitness).

- La congestion est un problème important en GP :
  - Perte de ressources mémoire
  - Perte de temps de calcul
  - Mauvaise lisibilité des solutions
- Les introns ne contribuent pas au fitness, mais ...
  - ils peuvent servir de réservoir de symboles pour la création future de sous-arbres intéressants
  - ils pourraient constituer des barrières pour atténuer les effets destructeurs du cross-over. Ce serait la raison de leur multiplication lorsque le fitness stagne : les bonnes solutions longues (avec des introns) auraient un avantage adaptatif sur les bonnes solutions courtes, plus fragiles.

# Lutte contre la congestion

- Utiliser des pénalités sur la longueur des programmes (Iba *et al.* 1994). Critères MDL, SRM (Teytaud *et al.* 2005)
- Utiliser un fitness multi-objectif Pareto : augmenter le fitness, diminuer la longueur (S. Luke)
- Simplifier les expressions, par exemple à l'aide de règles de substitution (ex : Eckart 1999).

# Accélération de la PG

- Emploi de code machine (Nordin, Banzhaf, 1994)
- Parallélisation/distribution : modèles en îlots, ... (voir AGs)
- Utilisation du GPU : voir école Yravals 2007.
- Utiliser seulement une partie du jeu de cas de fitness : Dynamic Subset Selection (Gathercole *et al.*, 1997).
- Astuces d'implémentation : pour  $n$  fitness cases, décoder  $n$  fois un arbre prend du temps  
→ retourner un vecteur avec un résultat par fitness case, à chaque noeud de l'arbre
- Et aussi lutte contre la congestion (voir suite).

- Livres :

- « Genetic Programming I,II,III,IV », 1992, 1994, 1998,2003, John Koza et al.
- « Genetic Programming: an introduction », 1998, Banzhaf et al.
- « Advances in Genetic Programming I,II,III», 1994 Kinear, 1996 & 1998, Angeline et al.
- « Genetic Programming and Data Structures », W.B. Langdon, 1998
- « Genetic Algorithms - principles & perspectives », C. Reeves, J. Rowe, 2003
- « Machine Learning », T. Mitchell, 1996,
- « Genetic Algorithms + Data Structures = Evolution Programs », Z. Michalewicz, 1999
- « An introduction to genetic algorithm », M. Mitchell, 1996
- « Multi-Objective Optimization Using Evolutionary Algorithms», 2001, K. Deb
- « Data Mining and Knowledge Discovery With Evolutionary Algorithms », 2002, A. Freitas

- Conférences
  - GECCO (Genetic Programming)
  - EuroGP
  - PPSN
  - Artificial Evolution
- Revues
  - IEEE Transactions on Evolutionary Computation
  - Genetic Programming and Evolvable Machines
  - Applied Soft Computing

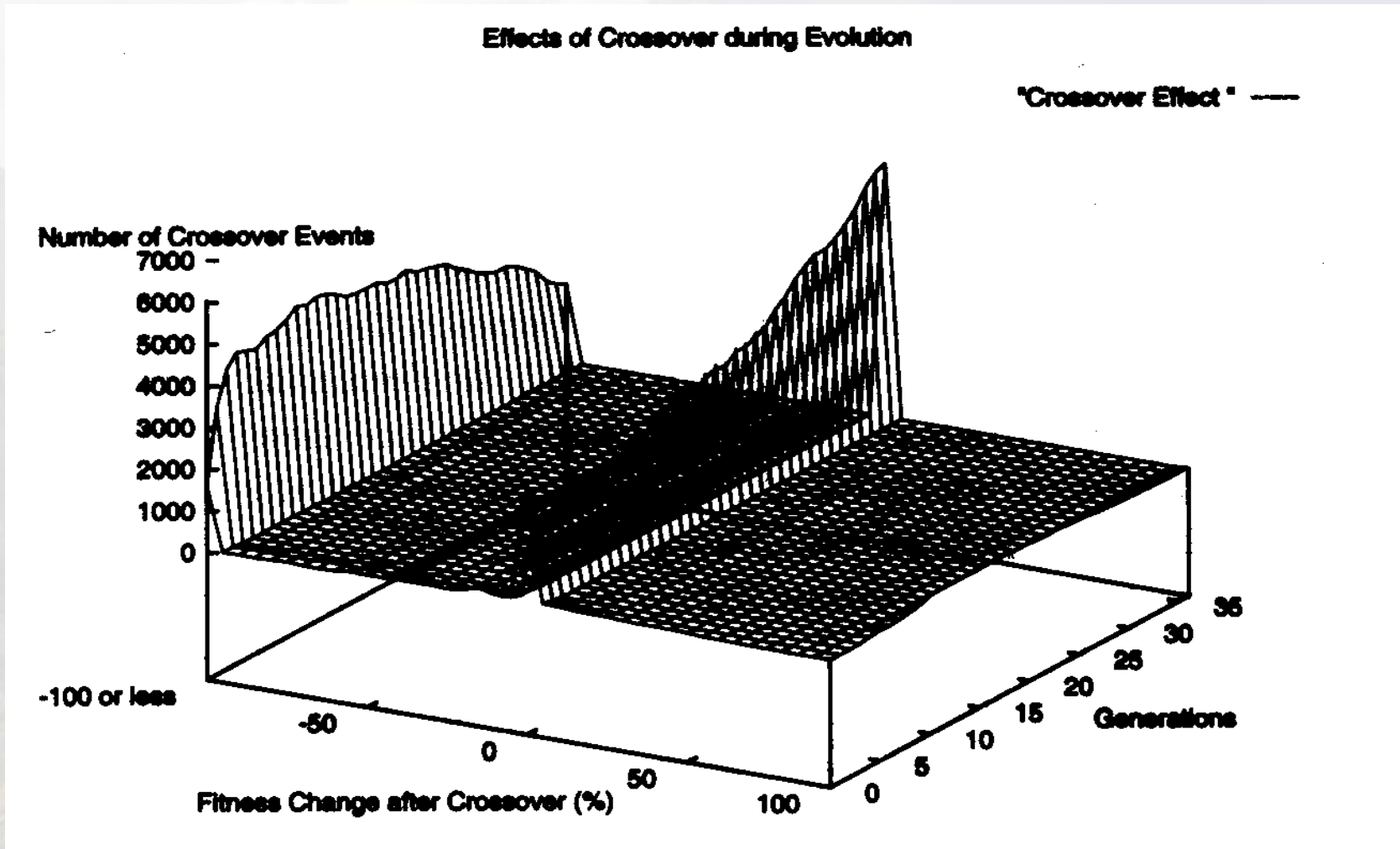
- Logiciels :
  - Koza (Lisp) : <ftp://ftp.io.com/pub/genetic-programming/code/koza-book-gp-implementation.lisp>
  - LIL-GP (C / C++) : <http://garage.cp.msu.edu/software/lil-gp>
  - EO (C++) : <http://eodev.sourceforge.net>
  - Open-BEAGLE (C++) : [http://www.gel.ulaval.ca/~beagle/index\\_f.html](http://www.gel.ulaval.ca/~beagle/index_f.html)
  - ECJ (Java) : <http://www.cs.umd.edu/projects/plus/ec/ecj/>
  - PerlGP (Perl) : <http://www.perlgp.org>
- Mailing list :
  - subscribe genetic programming
  - to
  - [genetic-programming-REQUEST@cs.stanford.edu](mailto:genetic-programming-REQUEST@cs.stanford.edu)

# Schémas pour la PG

- Th. des schémas pour la PG (2001, R. Poli)
  - première formulation exacte du th. des schémas pour la PG, valide pour le crossover standard;
  - la formulation demeure complexe (voir proceedings of EuroGP'2001)
  - « spécialisations » possibles, par exemple calcul de la taille moyenne des programmes en gén.  $t+1$ , en l'absence de mutation.
- Calcul de la distribution de la taille des arbres (R. Poli, EuroGP'07)
  - Présence d'un biais structurel
  - Sur-échantillonnage des petits arbres ?

# Rôle du cross-over en GP ?

- Cross-over constructif ou destructeur ? Une expérience de Banzhaf en PG "à la Koza" :



# Remplacer le cross-over ?

- Cross-over = macro-mutation ? Si oui, alors oublier le dogme de la PG « standard » sur le très faible taux de mutation. Controverse sur le sujet entre Koza (1992, 1995) d'une part, Angeline (1997), Luke et Spector (1997) d'autre part.
- Ouverture vers d'autres systèmes, exemple PIPE de Schmidhuber et Salustowicz (1997), où les arbres de la population sont générés à l'aide d'une table de probabilités, à la manière de la stratégie PBIL de Balujah pour les AGs : voir **Algos à Estimation de Distribution**
- Note : on peut concevoir d'autres formes de cross-over, moins destructrices, par exemple pour les problèmes d'optimisation paramétrique.