Programmation Génétique et Intelligence Artificielle

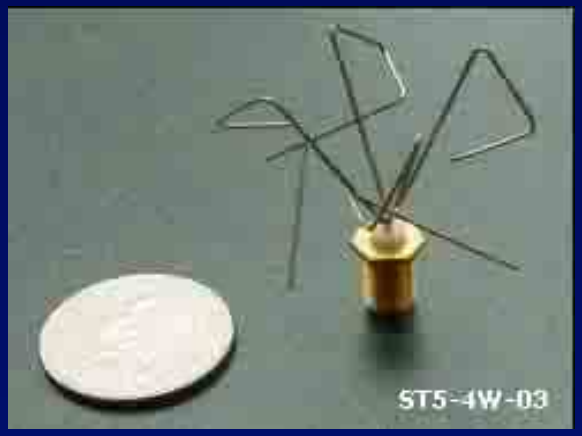
Pr Pierre Collet

Laboratoire des Sciences de l'Image, de l'Informatique et de la Télédétection
Equipe Fouille de Données et Bioinformatique Théorique
Université de Strasbourg

Pierre.Collet@lsiit.u-strasbg.fr

Optimisation *vs* Intelligence Artificielle

- But de l'IA traditionnelle : reconstruire une intelligence « humaine » dans une machine (test de Turing).
- Mais n'y a-t-il pas d'autres formes d'intelligence ?
- Problème de la Nasa : transmission de données vers la terre.



Antenne de satellite utilisée par la NASA,
conçue par un ordinateur

Qu'est-ce que la Programmation Génétique ?

C'est une branche « nouvelle » de l'évolution artificielle :

- EA : Algorithmes Génétiques et Stratégies d'Evolution
  - Font évoluer des tableaux de bits ou de réels de longueur fixe
  - Le génome est passé à une fonction d'évaluation comme une liste de paramètres
- Programmation génétique :
  - But initial : faire évoluer des programmes.
  - Fait évoluer des ***génomes de longueur*** **variable**,
  - ***On exécute l'individu*** pour trouver sa fitness.
- La PG est utilisée pour faire évoluer :
  - Des fonctions (régression symbolique)
  - Des constructions (structures et formes, circuits analogiques, lentilles, antennes, …)
  - Des programmes et des contrôleurs.

Historique Ordinateurs / EA / IA

- 1944 : ENIAC (Electronic Numerical Integrator And Calculator) (grosse machine à calculer, programme en dur).
- 1949 : Premier « vrai » ordinateur (architecture de von Neumann) : l'EDSAC (Electronic Delay Storage Automatic Calculator).
- 1950 : A. Turing écrit les premiers papiers sur ce que devraient être des ordinateurs « intelligents » (Computer Machinery and Intelligence, introduisant son « test de Turing »)
- 1950 : Shannon publie une analyse détaillée du jeu d'échecs vu comme un problème de recherche de solution.
- 1953 : Premier ordinateur commercial (IBM650).

  Premiers tests d'évolution artificielle aux US, en Australie et en Europe.

Historique IA/EA

- 1955 : « logic theorist » par Newell et Simon : en représentant un problème comme un arbre, le programme cherche la branche qui aurait le plus de chances de contenir la bonne solution.
- 1956 : McCarthy invite des collègues pour le « Dartmouth Summer Research Project on *Artificial Intelligence* ».
- 1957: Fraser fait évoluer des chaînes de bits par croisement.
- 1957 : General Problem Solver , par Newell, Shaw et Simon,
- 1958: Friedberg suggère que des ordinateurs pourraient s'« auto-programmer » par mutations successives de leur code.
- 1958 : McCarthy invente le LISP
- 1959: Friedmann écrit un article sur la manière dont l'évolution pourrait être artificiellement simulée.

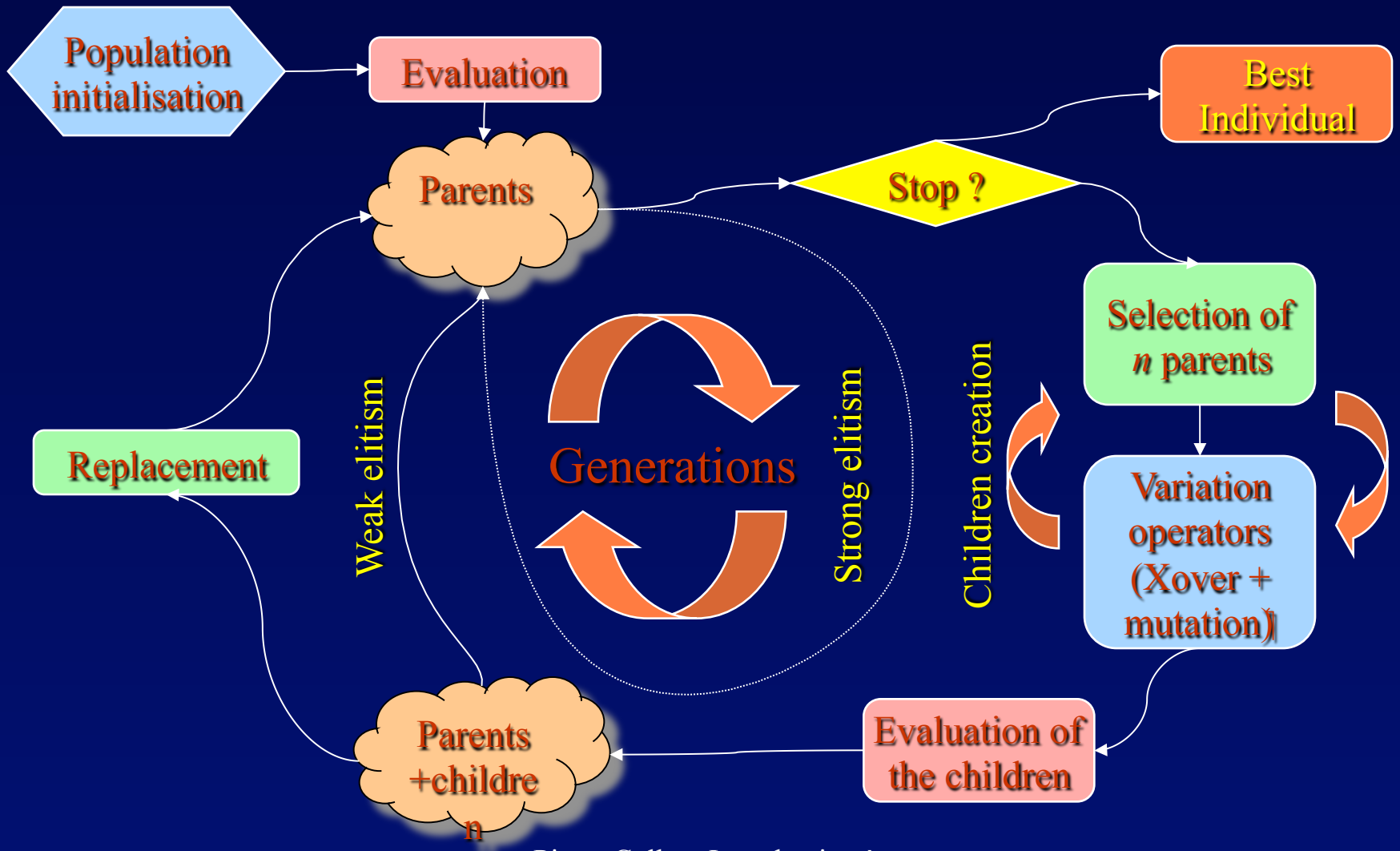
Historique de la PG

- 1958: Friedberg, essaye d'écrire un programme qui s'auto-modifie par mutations.
- 1980: Smith introduit des petits programmes dans les règles d'un système de classeurs (Learning Classifier System).
- 1985: Petit article de Cramer contenant (presque) tout :
  - Structure de taille variable en forme d'arbre pour représenter un programme.
  - Mutation, Xover, mais aussi *inversion*.
- 1987: Koza commence à travailler sur la Programmation Génétique
  - 1ers papiers intéressants en 1989 (problèmes de centrage de chariot, d'équilibre de balais),
  - 1992: Genetic Programming I.
  - 1994: Genetic Programming II (introduction des ADFs).
  - 1999: Genetic Programming III (évolution de structures)
  - 2003: Genetic Programming IV (obtention régulière de résultats compétitifs avec l'intelligence humaine)

Autres paradigmes de PG

- Linear Genetic Programming (Brameier[94], Nordin[94])
  - Liste d'instructions (possiblement de l'assembleur)
  - Base de Discipulus (logiciel de PG commercial)
- PADO (Teller, Veleoso [96]) Parallel Algorithm Discovery Orchestration
  - Programmation génétique à base de graphes
- Grammatical Evolution (O'Neil, Ryan [98])
  - Utilisation d'une grammaire BNF, où un génome est une liste d'entiers, et où à chaque entier est associé une règle appliquée avec un modulo sur l'ensemble des règles.
- Cartesian GP (Miller [00])
  - PG à base de graphe, où les nœuds sont repérés par leurs coordonnées cartésiennes
- Push-GP (Spector [01])
  - GP à piles
- …

Forme générale des AEs



Présentation de la PG à la Koza

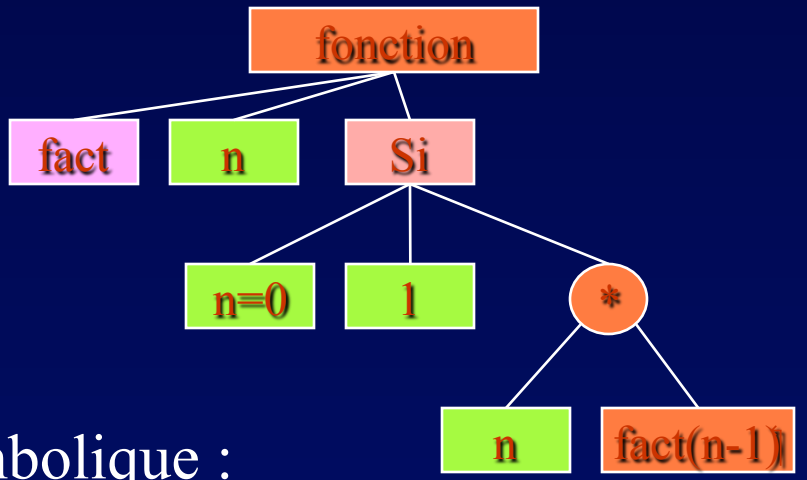
Algo de base : (Genetic Programming I "On the Programming of Computers by means of natural selection" MIT Press, 1992).

Représentation d'un individu :

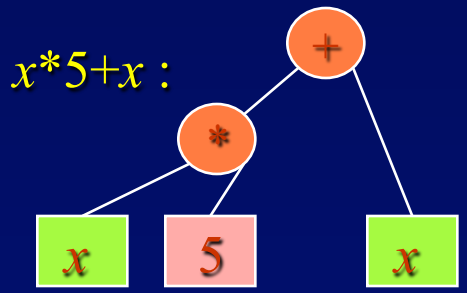
- Un individu est un arbre, représentant une fonction ou un programme fonctionnel
- L'utilisation d'un langage fonctionnel (LISP) réduit l'espace de recherche :
  - Pas d'effets de bord.
  - Tous les programmes sont valides.
- Cependant, itérations, boucles, appel de sous-programmes, récursion, … sont difficiles à implémenter.

Fonction LISP et régression symb.

- Fonction fact(n)
  Si n=0 renvoyer 1
  Sinon, renvoyer n*fact(n-1)



- Application à la régression symbolique :

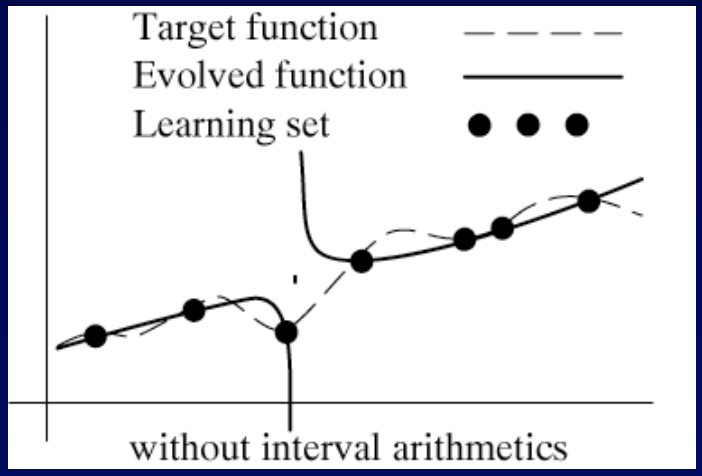


Terminaux et fonctions à la Koza

- Terminaux
  - ERC (Ephemeral Random Constants), habituellement dans [0,1[.
  - Actions (tourner_à_droite, avancer)
  - Variables indépendentes (entrées du pb : x, y, …)
- Ensemble de fonctions
  - Ensemble de fonctions : F={+,-,*,/,sin,…,if_food_ahead,…}
  - + appel à des ADFs (Automatically Defined Functions).
- Choix des fonctions :
  - Peut-on approximer $1/x$ avec {+,*} ?
  - avec {+,-,*,/,sin,cos,tan,exp,log,cosh, sinh, tanh,…} ?
  - On peut déterminer des familles de fonctions (T. Soule) en montrant que sin, cos, tan sont équivalents. Dans ce cas, 1 seul représentant d'une famille est mieux que l'ensemble des représentants.

Arité, domaine de définition

- Attention de bien gérer l'arité des opérateurs
- Résultats non définis : / avec 0 au dénominateur.
  - Mauvaise solution : protégér la division.



- Meilleure solution : Interval Arithmetics (M. Keijzer). On associe un intervalle à chaque nœud :
  - Pour x : [0,1[.
  - Pour +, $[fg_{inf}+fd_{inf}, fg_{sup}+fd_{sup}]$
  - Pour *, $[\min(fg_{inf}*fd_{inf}, fg_{inf}*fd_{sup}, fg_{sup}*fd_{inf}, fg_{sup}*fd_{sup}),$ $\max(fg_{inf}*fd_{inf}, fg_{inf}*fd_{sup}, fg_{sup}*fd_{inf}, fg_{sup}*fd_{sup})]$
  - Pour …

Initialisation des individus

- But : créer les meilleurs arbres possibles favorisant l'évolution des individus et en évitant convergence prématurée et bloat.
- Nombreuses méthodes proposées sans qu'aucune ne se détache réellement. D'où l'utilisation de Koza[92] :
  - Full (*t*) : *t* profondeur max autorisée. Tant que *t* non atteinte, full choisit des fonctions, sinon des terminaux.
  - Grow (*t*) : Tant que *t* non atteinte, grow choisit des fonctions ou terminaux, sinon terminaux seulement.
  - Ramped half and half : Pour augmenter la diversité, créer autant d'arbres par full ou par grow, avec une profondeur minimale de 2.
- Dans GP III : profondeur initiale maximale= 6.
- Dans GP III & IV, « developmental GP » : tous les individus partent d'un embryon fait main.

Fonction d'évaluation

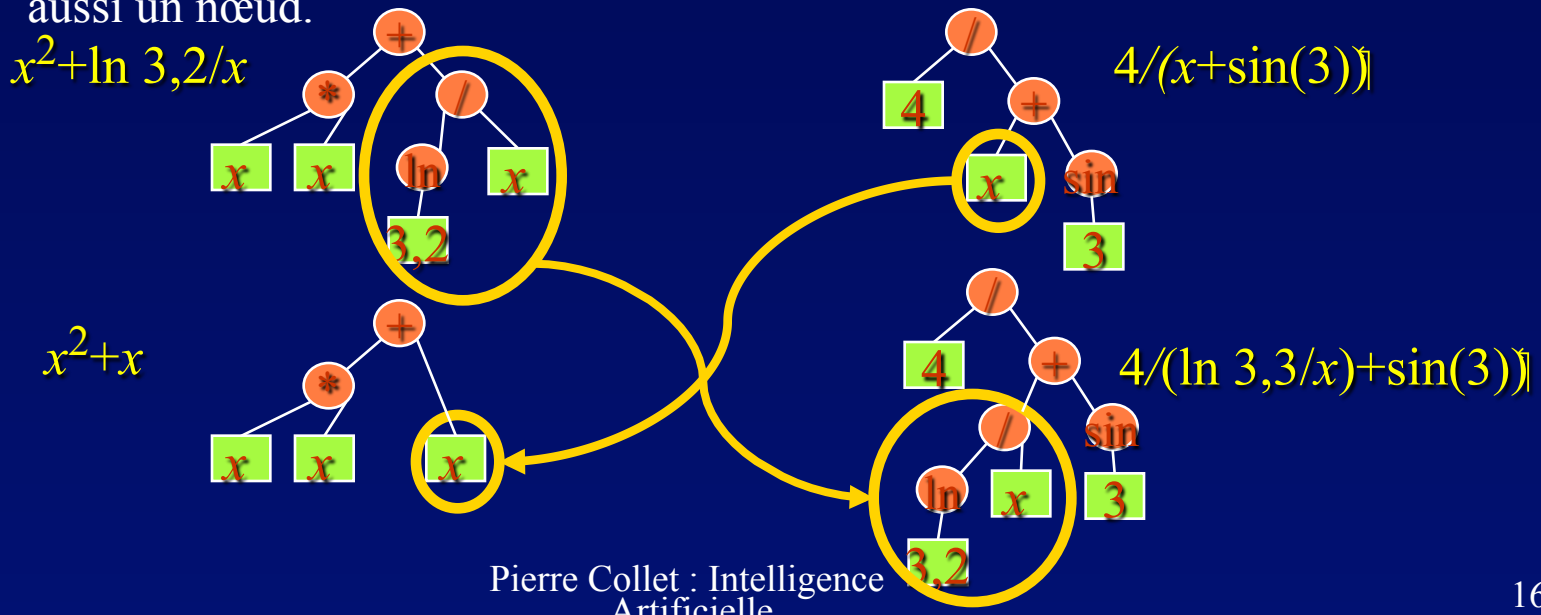
- Essentielle (comme d'hab !) : doit guider l'algo.
- Maximiser les points de nourriture trouvés en un nombre de pas maximum (fourmi artificielle), maximiser le nombre de points dans une cible (ifs inverse), …
- Si régression symbolique les fonctions usuelles à minimiser sont :
  - Ecart type : $$\mathrm{MSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(f(x_i)-s_i)^2}$$
  - Ecart type relatif : $$\mathrm{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\frac{f(x_i)-s_i}{s_i}\right)^2}$$

Sélection de parents d'après la fitness

- *Ad libitum*: dépend du problème à résoudre (tournoi, ranking, …).
- Dans GP III et IV : tournoi de taille 7 (cf. Goldberg et Deb, 1991).

Croisement

- Koza utilise d'hénaurmes populations (jusqu'à 1 million d'individus), et dit que le Xover est le principal opérateur de l'évolution. Dans GP I et II, Xover seul. Mais certains travaux ont montré l'intérêt de la mutation, donc dans GP III et IV, 1% de mutation.
- Croisement standard Koza : échange de sous-arbres en prenant un nœud avec une proba de 90%. GP I et II 2 parents donnent 2 enfants. GP III & IV, 2 parents donnent 1 enfant.
- GP III : Si le locus du parent 1 est un nœud, alors le locus du parent 2 est aussi un nœud.



Autres croisements

- Height-Fair Xover (O'Reilly, Oppacher[94]) : Proba plus grande d'avoir un croisement proche des feuilles (d'où le 90% de Koza). Proposition de sélectionner d'abord une profondeur sélectionnée, puis de choisir un nœud aléatoirement dans la profondeur choisie.
- GP-Std / Same (Harries, Smith[97]) : échanger des sous-arbres de la même profondeur dans les 2 parents. Résultat pires que le croisement standard… sauf si utilisé avec 1 croisement standard à 50% (d'où le nom).
- Homologous Xover : on choisit un ss-arbre dans parent1 (avec 90% chances pour un nœud) et dans parent2, on cherche un sous-arbre de structure et position similaire. Si ça ne fonctionne pas comme prévu (résultats identiques à un croisement standard), permet de lutter efficacement contre le *bloat*.
- …

Mutation

- En PG, habituellement appliquée entre 0 et 1%.
  - Mutation standard : supprimer un ss-arbre et le remplacer par un nouveau ss-arbre. Suggestion de Koza : sélectionner un nœud à 90%.
  - Small mutation (mutation minimale) : si mutation d'un opérateur, on remplace par un autre opérateur de même arité. Si terminal, on remplace par un autre terminal. Si terminal numérique, on ajoute un bruit gaussien.
  - Croisement monoparental : échange de 2 sous-arbres dans un même parent.
- Small mutation et croisement monoparental sont trop doux (ne permettent pas de sortir d'un minimum local) mais améliorent les résultats si utilisés en conjonction avec la mutation standard.

Notion d'*intron*

- En 1977, découverte que 90% du code génétique contenu dans l'ADN n'est pas exprimé !
- Génome composé de courtes séquences codantes (exons) entrecoupées de longues séquences éliminées par des enzymes (introns).
- Les introns sont structurés (contiennent de l'info), mais sont supprimés par les enzymes.
- En PG, lors d'une « soft » mutation, possibilité de remplacer une fonction par une autre d'arité différente. Si arité supérieure, on crée un sous-arbre. Si arité inférieure, que faire du sous-arbre surnuméraire ? (suppression = gâchis $\Rightarrow$ intron).

Remplacement (= sélection des survivants)

- Traditionnellement, 2 remplacements utilisés : steady-state et générationnel.
- Mais apparemment (Syswerda [91]), aucune raison de se limiter à ces deux techniques : utilisation de $(\mu+\lambda)$ ou (μ,λ) possible.
- Méthode de remplacement choisie d'après le pb à résoudre:
  - Pour évoluer un contrôleur de robots, le steady-state est idéal (Nordin & Banzhaf [95]).

Taille de la population

- A nb d'évaluations identiques, vaut-il mieux une grande population et peu de générations ou petite population et beaucoup de générations ?
- Grande population = ralentissement de la convergence, or la PG est connue pour converger rapidement.
- Les expériences de Koza montrent que plus la population est grande, meilleurs sont les résultats.
- Koza utilise des populations >> 1000, avec un nombre de générations < 100.
- Mais Luke [98] montre qu'en fait, ça dépend surtout du problème à résoudre (comme d'hab).

Gros pb en PG : le *bloat*

- Pb touchant tous les algos évolutionnaires à taille d'individu variable : évolution normale pendant un certain temps, puis tout d'un coup, croissance démesurée de tous les individus.
  - Du fait de la structure arborescente, en quelques générations, l'espace mémoire occupé et le temps d'exécution explosent.
- Explication avancée : plus le ratio exons/introns est faible, moins les opérateurs génétiques ont de chance de dégrader les individus. A un moment, l'évolution s'en rend compte et favorise les gros individus.

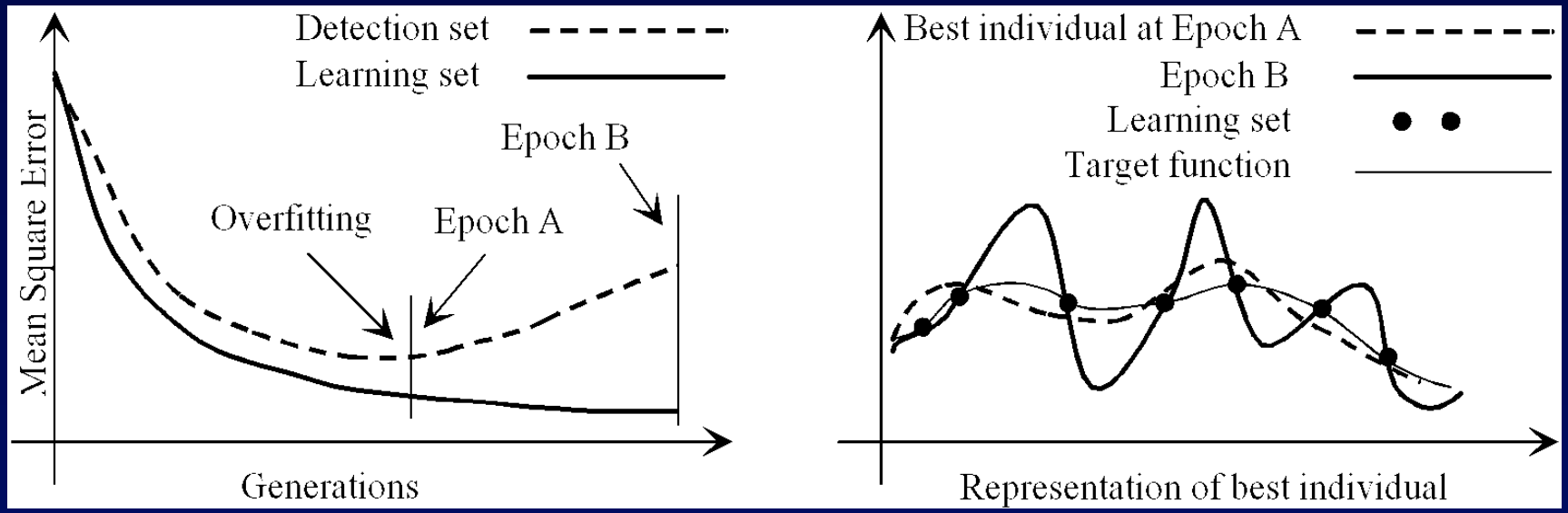
Lutte contre le bloat (parcimonie)

- Solution de Koza : profondeur maximale de 17 ! (GP I).
- Tarpeian GP (Poli [03]) ne pas évaluer les individus au delà d'un nombre de nœuds maximum : leur donner une mauvaise fitness sans les évaluer.
- Plus intéressant : Soule[03] émet l'hypothèse que le bloat provient des opérateurs génétiques.
  - Solution proposée : appeler l'opérateur de croisement avec une probabilité par nœud (comme pour la mutation) et pas 1 fois par individu. Ainsi, le rapport exons/introns ne protège plus les gros individus.
  - C'est une pénalité aux gros individus, mais pénalité constructive (application de plusieurs croisements).

Sur-apprentissage

- Observation : plus un individu est petit, plus son résultat est généralisable.
- Au début de l'évolution, amélioration des résultats avec des petits individus (moins de choses à optimiser) jusqu'à-ce qu'on arrive à un seuil où les améliorations génériques ne sont plus possibles.
- Alors, on rentre dans une phase où l'amélioration provient de l'ajout de verrues aux individus génériques pour mieux approximer un point précis.
- Le bloat serait une expression du sur-apprentissage ?
- Malheureusement, contrôler le bloat empêche souvent de trouver de bons individus… $\Rightarrow$ il faut voir au cas par cas !

Détection du sur-apprentissage et évaluation



Nécessité d'avoir 3 jeux de données (Schoenauer[06]):

- 1 jeu d'apprentissage (sur lequel on fait évoluer les individus)
- 1 jeu de test (pour détecter le surapprentissage).
- 1 jeu d'évaluation (pour évaluer l'individu sur un jeu qui n'a pas participé à l'évolution de l'individu).

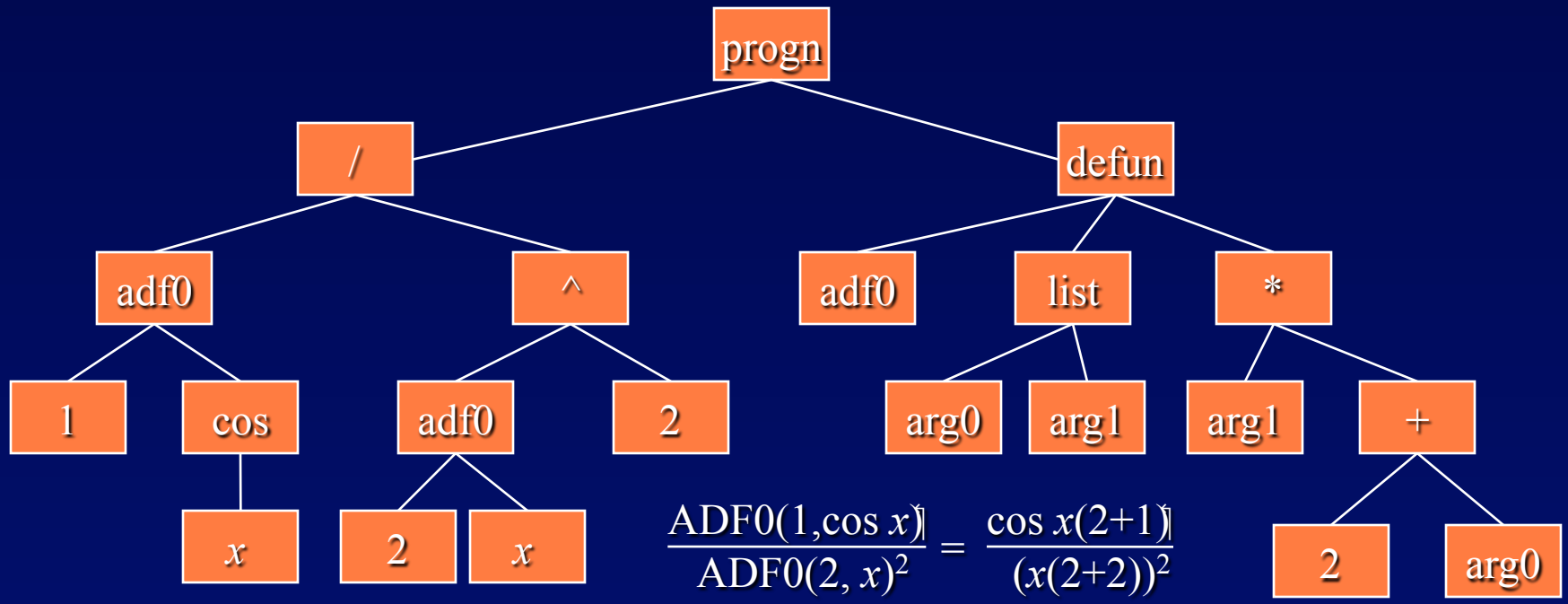
C. Gagné, M. Schoenauer & al., « GP, Validation Sets and Parsimony Pressure », EuroGP'06

Advanced Genetic Programming *à la* Koza

- Genetic Programming II (*Automatic discovery of Reusable Programs*) MIT Press, 1994:
  - Introduction of ADFs
- Genetic Programming III (*Darwinian Invention and Problem Solving*) Morgan Kaufmann, 1999.
  - ADFs and Architecture Altering Operations
- Genetic Programming IV (*Routine Human-Competitive Machine Intelligence*) Kluwer, 2003.
  - Human competitive, patent infringing and patentable results

Automatically Defined Functions

Si le programmeur pense que le problème pourrait réutiliser certaines parties plusieurs fois, définir une ou plusieurs ADF.

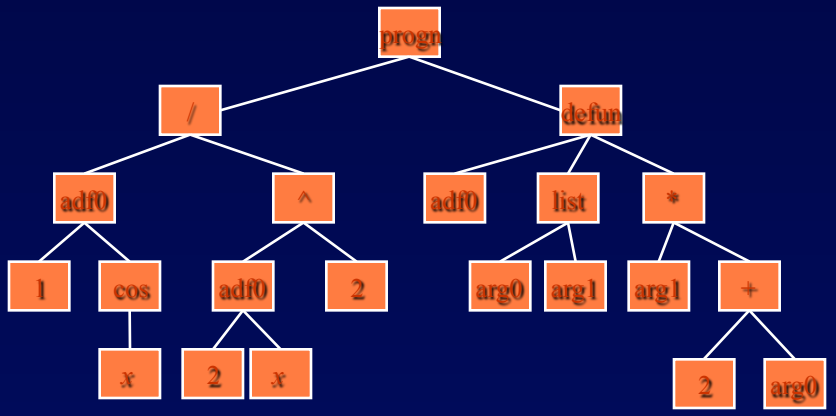


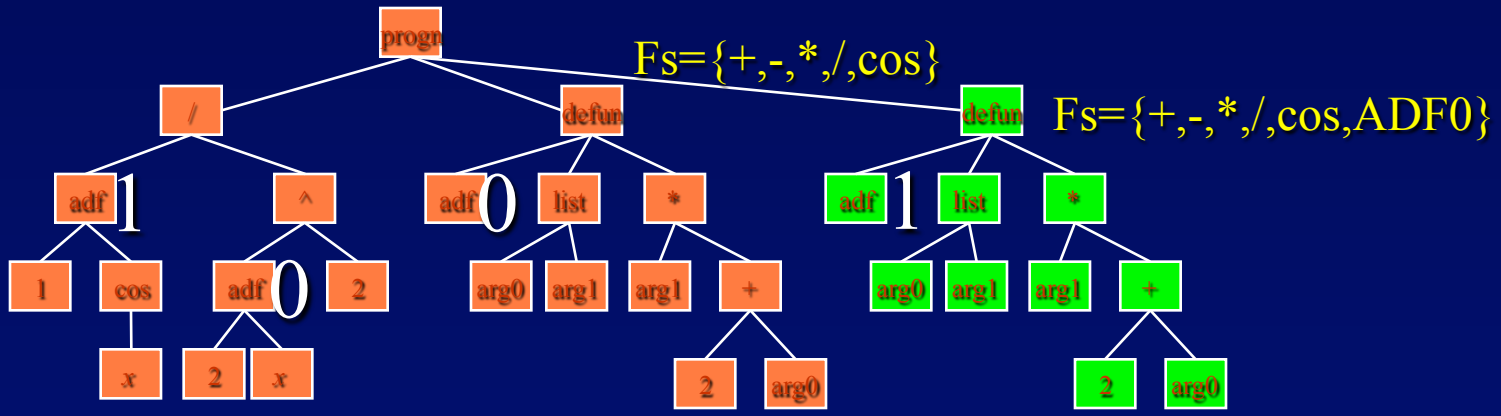
Si définition de plusieurs AFDs (ADF0, ADF1, …), le jeu de fonctions de ADF*x* peut contenir des appels à ADFy ssi y<x *(pour éviter une récursion involontaire et incontrôlée).*

Duplication de sous-programme

- Idée d'origine: thèse (biologie) de Ohno's (1970) disant que les changements évolutifs majeurs apparaissent par duplication et délétion de gènes dans la nature.
- A chaque génération, décider avec *prob_sub_dupl* si on doit utiliser la duplication de sous-programme sur un individu.
- Choisir un individu parmi les meilleurs (opérateur de sélection), et le dupliquer (pour conserver une copie de l'original).
- Si son nombre d'ADFs est > *max_adfs*, ne rien faire.
- Sinon, prendre un ADF*x* dans l'individu, le dupliquer et le renommer ADF*n+1*.
- Ajouter ADF*n* au jeu de fonctions de ADF*n+1*.
- Partout où ADF*x* était appelé, appeler aléatoirement ADFx ou ADFn +1 avec probabilité égale.

Subroutine duplication

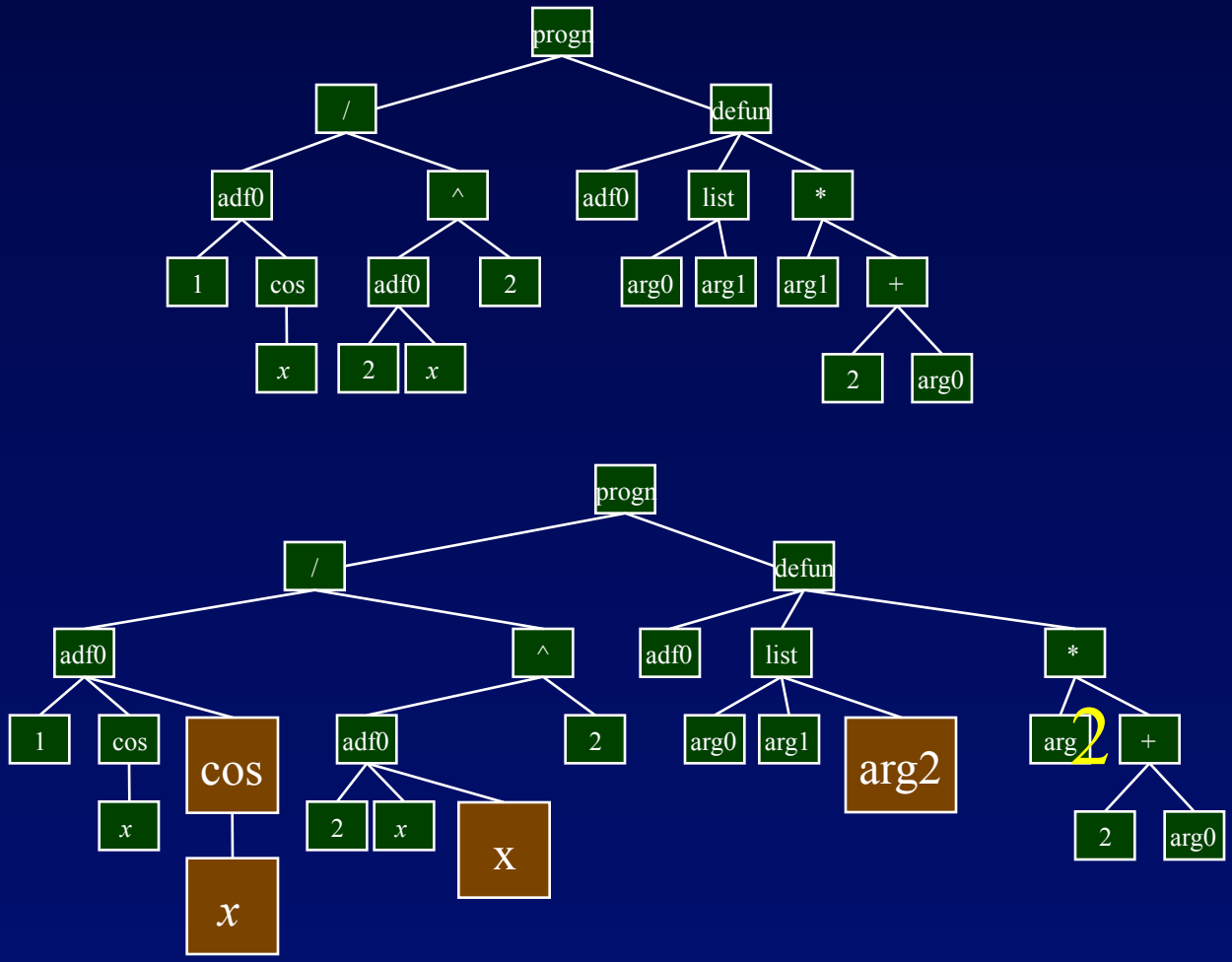




Argument duplication

- At each generation, decide with *prob_arg_dupl* if the subroutine duplication operator is to be used on an individual.
- Choose an individual among the best (selection operator), duplicate it so as to keep a copy of the original individual.
- Pick one of the ADFs of the individual. If the number of arguments *n* is greater than *max_args,* don't do anything.
- Else, pick an argument arg*x* in the ADF, create a new argument arg*n+1,* and add 1 to the arity of the ADF.
- Wherever arg*x* was called anywhere in the ADF, randomly use arg*x* or arg*n+1* with equal probability.
- Wherever the ADF, was used in the individual, duplicate the subtree of the x^{th} argument and add it as the $n+1^{th}$ argument of the ADF call.

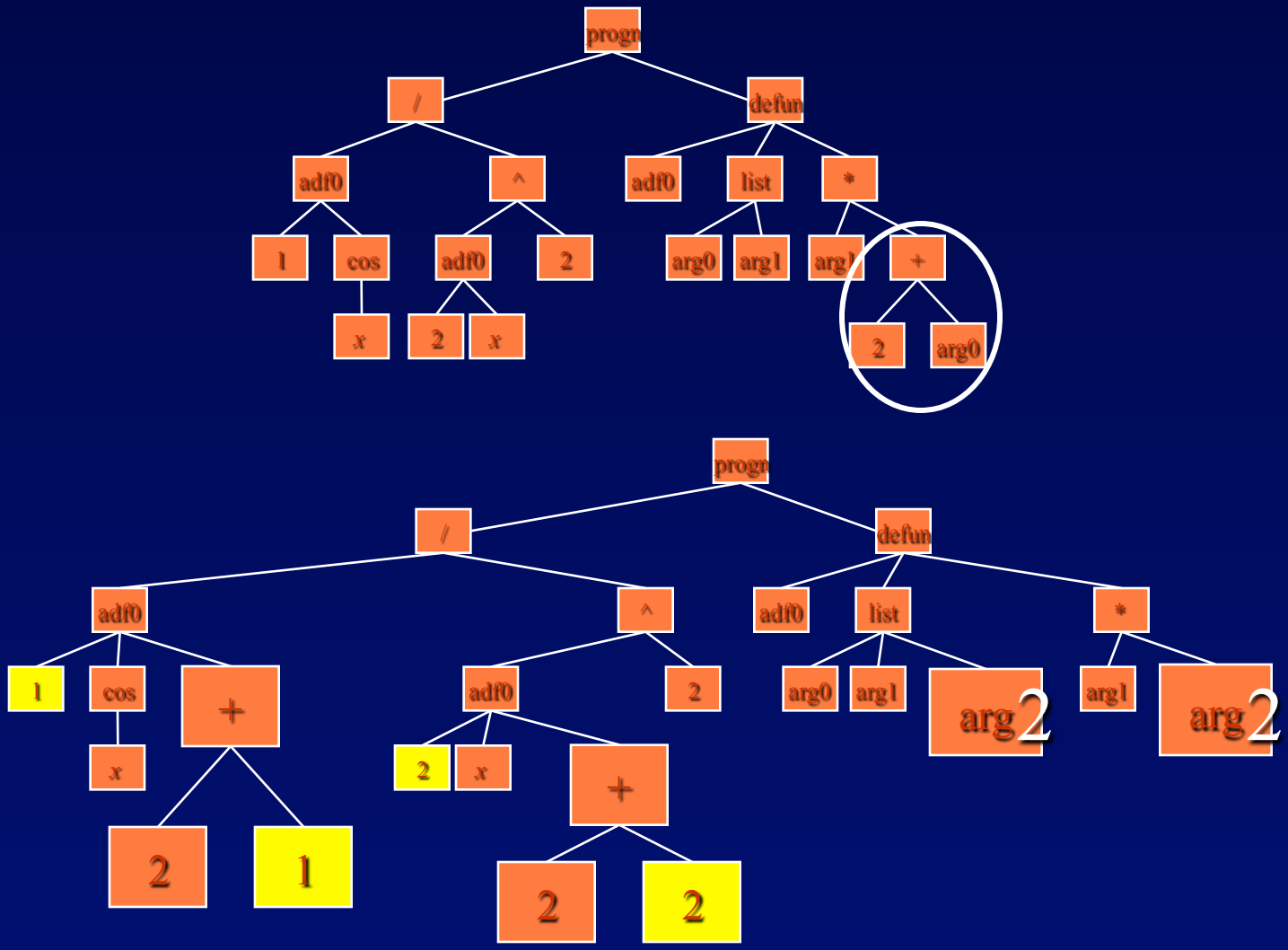
Argument duplication



Argument creation

- At each generation, decide with *prob_arg_creat* if the argument creation operator is to be used on an individual.
- Choose an individual among the best (selection operator), duplicate it so as to keep a copy of the original individual.
- Pick an ADF*x* in the individual. If its number of arguments *n* is ≥ *max_args,* don't do anything.
- Else, add an argument *n+1* to the ADF.
- Pick a subtree in ADF*x* to turn into argument *n+1*.
- Memorize the subtree, and replace it by ARG*n+1*.
- Traverse the individual, and wherever there is a call to ADF*x*, add the memorized subtree as argument *n+1*.

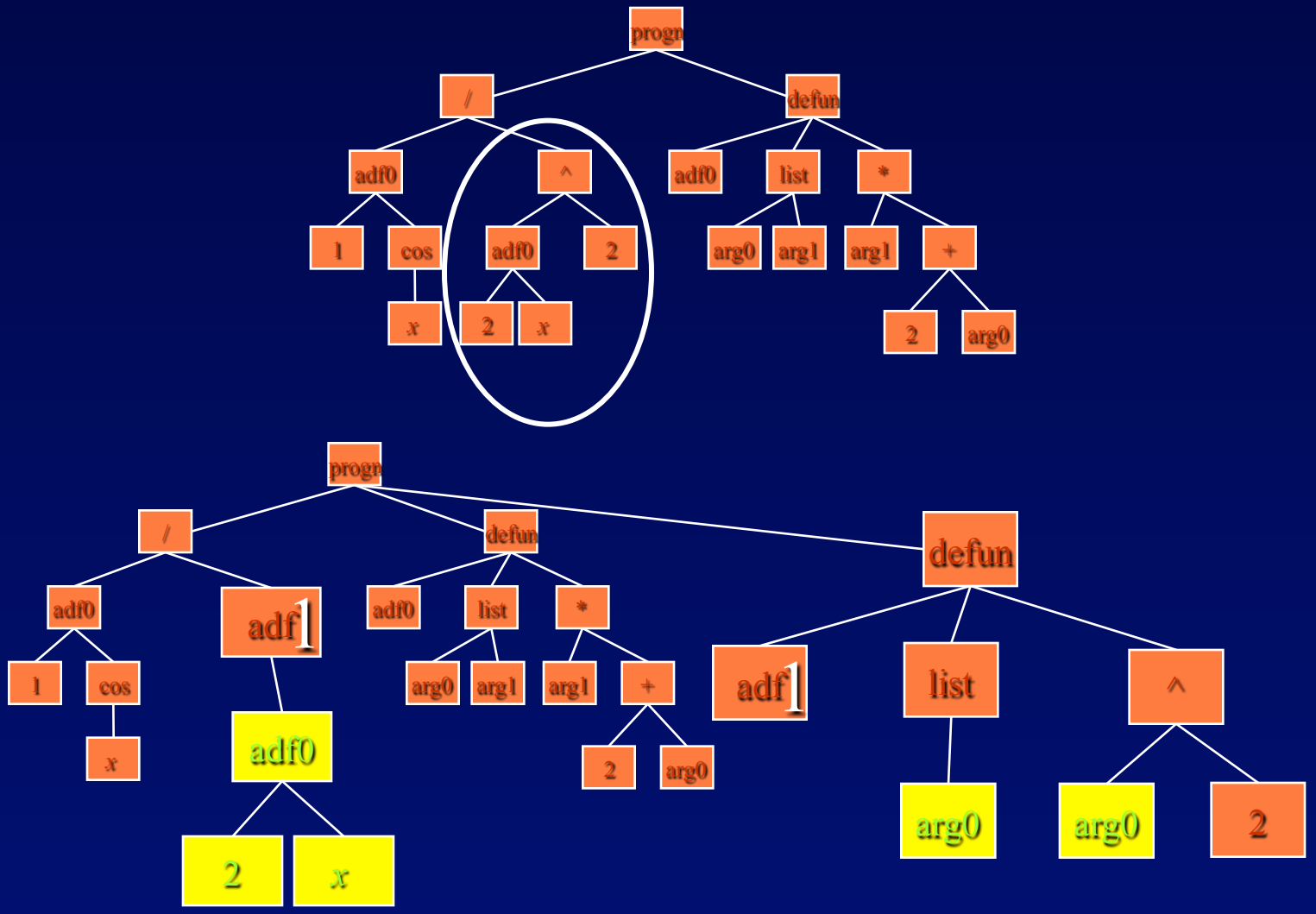
Argument creation



Subroutine creation

- At each generation, decide with *prob_sub_creat* if the subroutine creation operator is to be used on an individual.
- Choose an individual among the best (selection operator), duplicate it so as to keep a copy of the original individual.
- If its number of ADFs *n* is greater than *max_adfs,* don't do anything.
- Else, pick a subtree and create a new ADF*n+1* with it, by traversing it depth first.
- For each encountered node, determine stochastically if its subtree should be an argument of the new ADF. If so, replace the subtree by an argument and add it in the list of arguments of the new ADF.
- Replace the picked subtree by a call to ADF*n+1*, using the deleted subtrees as arguments for the ADF.

Subroutine creation



Deletions

- Subroutines can also be deleted, with a variation of subtleties:
  - Delete by macro expansion: calls to the deleted ADF are replaced by copies of the ADF. This preserves the semantics of the programme, at the expense of an increase in size.
  - Delete by consolidation: calls to the deleted ADF are made to another ADF,
  - Delete with random generation: calls to the deleted ADF are replaced by randomly generated subtrees,
- Arguments can also be deleted (again by consolidation, random generation and macro expansion).
- Details to be found in Chapter 5 of Koza's Genetic Programming III.

Structure preserving Xover

- If ADFs and architecture altering operators are used, a subtree selected in Parent1 will possibly refer to ADFs that are not present in Parent2…
- 3 possibilities:
  - Branch typing: a separate type is assigned to each branch (ADFs and main function) depending on the functions and terminals set.
  - Like-branch typing: several branches sharing the same functions and terminals set are given the same type.
  - Point typing: a type is assigned to each node, depending on the contents of its subtree.
- When a Xover occurs, a first subtree is chosen at random in Parent1. Then, whether Branch, Like-branch or Point typing are used, the chosen subtree is inserted in a subtree of compatible type.
- Branch & Like-Branch typing are used in individuals sharing a common structure. Point typing is used if architecture altering operators are used.

Automatically Defined Iterations (ADIs)

- Iterations consist of four elements:
  - Initialisation, body, update step and termination condition.
- Infinite loops will occur, so cpu rationing counter-measures must be adopted, by imposing a time limit on the execution of each individual, for instance.
- Or… perform iteratively some steps on a preestablished sequence, vector or array = ADI (described in GP III chap. 6).
- A simplified version of the ADIs executes each ADI once. Then, when ADI0 is referred to in the evolved program, one substitutes the return value of the first execution of ADI0.
- Architecture altering operations (iteration creation, duplication, deletion, arguments creation, duplication and deletion) are described in Chap. 6.

Automatically Defined Loops (ADLs)

- 4 distinctive branches:
  - Loop Initialisation Branch (LIB),
  - Loop Condition Branch (LCB),
  - Loop Body Branch (LBB),
  - Loop Update Branch (LUB).
- When an ADL is called, the LIB and LCB are executed. If the LCB returns a positive value, the LBB is executed and then the LUB.
- Any of the 4 branches may contain referenced to other ADLs, meaning that nested iterative structures are possible (within a preestablished limit on the depth of the nesting).
- A simplified version of the ADL also exists, where each ADL is executed once before the main function is evaluated. Each reference to an ADL returns the value obtained by the last execution of the loop.
- Architecture altering operations for ADLs are provided in Chap. 7 (loop creation, duplication, deletion and loop arguments creation, duplication and deletion).

Automatically Defined Recursion (ADR)

- 4 branches:
  - Recursion Condition Branch (RCB),
  - Recursion Body Branch (RBB),
  - Recursion Update Branch (RUB),
  - Recursion Ground Branch (RGB).
- While the Recursion Condition Branch returns a positive value, the Recursion Body Branch is executed. The RBB may reference the ADF of which it is part. The RUB is then executed. When the execution is terminated (RCB returning a negative or null value), the Recursion Ground Branch is executed once. The return value of the ADR is that of the RGB.
- Architecture altering operators are defined on ADRs (recursion creation, duplication and deletion, recursion arguments creation, duplication and deletion).
- Cf. GP III Chap 8.

Automatically Defined Storage

- ADS is implemented by adding 2 new branches to a computer program:
  - a Storage Writing Branch (SWB) and
  - a Storage Reading Branch (SRB).
- 5 different types of internal storage (5 dimensions):
  - Dimension 0: Named memory, pushdown stack, queue.
  - Dimension 1: Indexed (vector) memory, list.
  - Dimension 2: 2 dimensional matrix, relational memory.
  - Dimension 3: 3 dimensional array.
  - Dimension 4: 4 dimensional array.
- Architecture altering operators are defined for ADSs (storage creation, deletion, duplication, dynamic changes in the dimensionality, dealt with by storage arguments duplication and deletion). Cf. GP III Chap 9, but also W. Langdon, *Genetic Programming + Data Structures = Automatic Programming*, Amsterdam, Kluwer, 1998.

Résultats personnels de Koza GP III / IV

Résultats présentés dans GP III et GP IV (2003) :

- 36 résultats « compétitifs avec l'intelligence humaine » obtenus par PG en 2003.
- Dans 23 cas obtenus par Koza, la PG a dupliqué la fonctionnalité d'une invention brevetée, ou a retrouvé des éléments d'un brevet, ou a créé une nouvelle invention brevetable.
- 15 cas (parmi les 23) concernent un brevet post-2000.
- Dans 6 cas, la PG a obtenu un résultat équivalent à une invention brevetée après le 1er janvier 2000.
- Dans 2 cas, la PG a créé une invention brevetable.

Critères de compétitivité avec l'intelligence humaine

8 critères de critères de compétitivité avec l'intelligence humaine ont été établis par la communauté évolutionnaire internationale :

A. Le résultat a été précédemment breveté comme une invention, une amélioration sur une invention brevetée, ou pourrait être breveté aujourd'hui.
B. Le résultat est équivalent ou supérieur à un résultat publié dans une revue scientifique reconnue à comité de lecture.
C. Le résultat est équivalent ou supérieur à un résultat placé dans une archive de meilleurs résultats connus, maintenue à jour par un panel d'experts scientifiques internationalement reconnus.
D. Le résultat est publiable de plein droit en tant que nouveau résultat scientifique, indépendemment du fait qu'il a été obtenu de manière mécanique.

Critères de compétitivité avec l'intelligence humaine

A. Le résultat est équivalent ou supérieur à la solution humaine la plus récente à un problème ancien, pour lequel une succession d'améliorations ont été élaborées par des humains.
B. Le résultat est équivalent ou supérieur à un résultat considéré comme une avancée dans son domaine au moment où il a été obtenu.
C. Le résultat résout un problème d'une difficulté reconnue dans son domaine.
D. Le résultat obtient un classement honorable, ou gagne une compétition humaine (où les joueurs sont soit des joueurs humains, soit des programmes écrits par des humains).

Résultats obtenus par PG par Koza

Cas des 6 résultats équivalents ou supérieurs à une invention brevetée après 2000. Les brevets ciblés ont été choisis dans ceux déposés entre janvier 2000 et juin 2001, pour inclure :

- des types de circuits différents (au moins un circuit analogique/numérique, au moins un circuit à entrées multiples, au moins un circuit évalué par chacune des métriques communes en électronique (analyse temporelle, analyse fréquentielle, analyse en courant continu, analyse de Fourier).
- Des circuits importants ou d'intérêt général.
- Des brevets où le comportement et les caractéristiques du circuit apparaissent de manière claire et quantitative.
- Des brevets comportant suffisamment d'information pour reproduire le circuit breveté en simulateur.

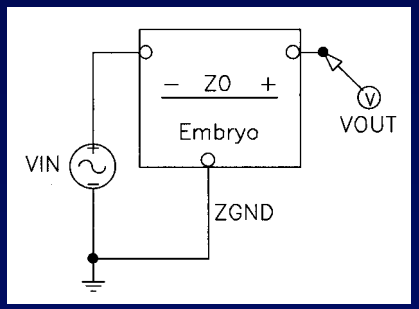
Différence avec résultats précédents de l'IA

- Koza introduit la notion du rapport A/I = rapport entre l'*Artificiel* par rapport à l'*Intelligence* apportée par l'homme dans l'obtention du résultat.
- Explication :
  - Un Système Expert exploite une base de règles établie avec un expert humain (ou par une série d'observations mises sous forme de règles par un arbre de décision, par exemple). Le SE n'exploite pas d'intelligence Artificielle, mais de l'Intelligence Humaine. Le rapport A/I est très faible.
  - En 1997, Deeper Blue « bat » Kasparov…, avec :
    - Des années de programmation, avec la participation de Miguel Illescas, John Fedorowicz, Nick De Firmian et Joel Benjamin (Grands maîtres).
    - Une BD de 700 000 parties de grands maîtres.

    Là aussi, très faible rapport A/I.

Etude du rapport A/I

- Pour Koza, le but ayant été de montrer le potentiel de la PG pour créer de l'intelligence artificielle, pour tous les problèmes l'algorithme de PG était ***absolument*** identique :
  - Initialisation de la population avec un embryon de circuit ***identique*** : un fil (c'est à la PG de trouver le reste).
  - Algorithme de Programmation Génétique ***identique***.
  - Jeu de fonctions et de terminaux ***identique***.
  - Paramètres de contrôle ***identiques***.
  - Critères d'arrêt ***identiques***.
  - Machine parallèle ***identique***, simulateur SPICE ***identique***.

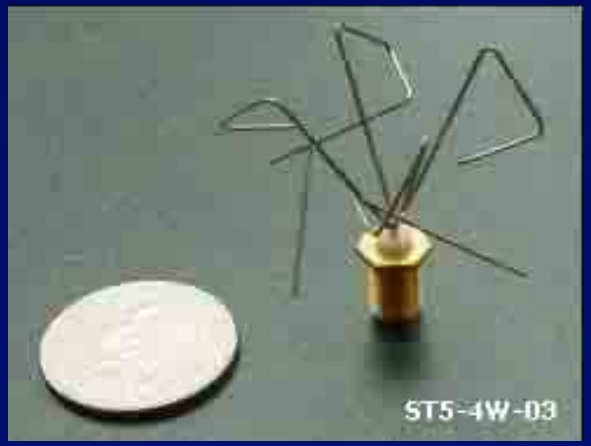


- Seule différence entre les problèmes résolus :
  - Fonction d'objectif différente (spécifique aux circuits ciblés).
  - Nombre d'entrées et de sorties (spécifique aux circuits ciblés).
  - La complexité des circuits étant très différente, la taille de la population a été réduite ou augmentée sur le seul critère d'obtenir un temps de calcul raisonnable.

Critique habituelles sur la PG

Ca ne fonctionne que sur des circuits, et que car Koza est devenu un spécialiste des circuits analogiques.

- Koza fait maintenant de l'optique géométrique, et obtient des résultats brevetables dans ce nouveau domaine (où il existe des simulateurs réalistes), en appuyant sur « RETURN ».
- De nombreux résultats compétitifs avec l'intelligence humaine ont été obtenus dans d'autre domaines, par exemple dans la conception d'antennes (cf. antenne obtenue pour la mission ST-5 de la Nasa, faisant bien mieux que les précédents résultats connus).



Domaines d'utilisation de la PG

- La PG est utilisable dès lors qu'on a défini un objectif à atteindre (fonction de fitness) et qu'il existe un simulateur dans le domaine du but recherché (pour accélérer le temps).
- Ce qui différencie la PG de l'EA, c'est que la structure du résultat est inconnue. Là où l'EA est utilisable dans des cas où il faut optimiser un jeu de paramètres, la PG contient les opérateurs permettant de faire évoluer la structure du résultat.
- Du coup, l'espace de recherche est très vaste, et la PG est très gourmande en temps de calcul. Koza est un précurseur notamment car il est milliardaire (ferme de 1000 Pentiums, à 400W/h le pentium + système de refroidissement = 0.5MW/h (!) sans parler de la maintenance, la construction d'un bâtiment adéquat, …)

Effort calculatoire (computational effort : Koza)

- Il est difficile d'évaluer la performance d'un algo stochastique, car :
  - Ce sont des algorithmes probabilistes : on ne peut pas garantir le succès à la génération G.
  - Quand un run ne réussit pas à trouver un bon résultat à la génération G, comment savoir s'il était sur la bonne voie de trouver un bon résultat ?
  - Comment dire au bout de combien de générations on va trouver le résultat escompté ?
- Computational effort :
  - Soit P(n) la probabilité de succès cumulée par nb d'évaluations (n) = nb de runs ayant réussi avant la $n^{ème}$ évaluation / nb runs total.
  - I(n,z) = nb d'évaluations à effectuer pour obtenir une solution valide avec une proba > z (généralement 99%) avec la formule :

$$\text{Computational Effort } I(n,z) = n * \left[\frac{\ln(1-z)}{\ln(1-P(n))} \right]$$

Observation sur la puissance de calcul…

- De manière intéressante, on peut établir une relation entre puissance de calcul et résultats obtenus :

  - Puissance de la première machine de Koza (pour livres GPI et II) : 0.00216 petacycles/jour (10^{15}). Bons résultats sur problèmes jouets.
  - Deuxième machine 9 x plus puissante, avec 0.02Pcyles/jour. Deux résultats compétitifs avec l'intelligence humaine (pas des brevets).
  - 3ème machine : 0.44 Pcycles/jour, avec premiers résultats retrouvant des inventions brevetées au 20è siècle.
  - 4ème machine : 3.2 Pcycles/jour, pour obtention régulière de résultats retrouvant des inventions brevetées au 20è siècle.
  - 5è machine (celle du livre de 2003) : 1000 Pentium II à 300Mhz, pour 30 Pcycles/jour, pour des résultats retrouvant des brevets du 21è siècle.
  - Faire tourner le cluster pendant 4 semaines ramène des résultats brevetables (= 1 journée de calcul à 840 Pcycles/jour).
- En 2005, un PC à 3Ghz fait du 0.3 Pcycles/jour.
- Si la puissance parallèle continue à doubler tous les 18 mois on arrive à 307 Pcycles/jour sur un portable dans 10 ans = 3 jours de calcul pour 1 brevet.

Salle J4 :

- 822MHz*384coeurs*2cartes*20machines*3600secondes*24heures = 10908794880000
- 1.10^{18} cycles/jour = 1000 Pcycles/jour.
- TSUBAME 2.0 :
  1442 PC
- 2 processeur hexacore
- 3 cartes GPU
- 448 cœurs par carte à 575MHz

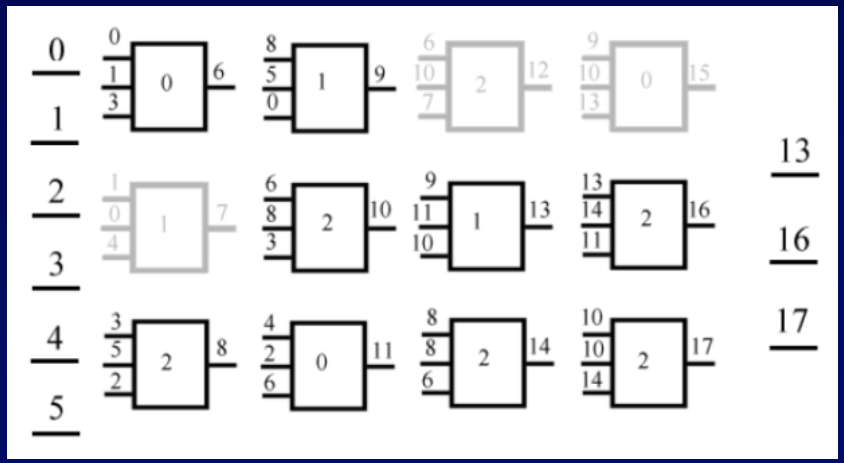
Conclusions

- Pour l'instant, Koza et ses super-ordinateurs obtiennent régulièrement des résultats brevetables sans apport d'intelligence humaine à la solution trouvée, mais de plus en plus de résultats brevetables ou compétitifs avec l'intelligence humaine sont trouvés par Programmation Génétique par d'autres équipes.
- Assistons-nous à l'apparition d'une première vraie instance d'intelligence artificielle ?
- Commentaire de Mc Carthy sur le bouquin de Koza :

  « John Koza's genetic programming approach to machine discovery can invent solutions to more complex specifications than any other I have seen. »

Cartesian GP (J. Miller, 2000)

- Cartesian : un « programme » est représenté comme une collection de nœuds sur une grille. Chaque nœud est identifié par ses coordonnées cartésiennes.

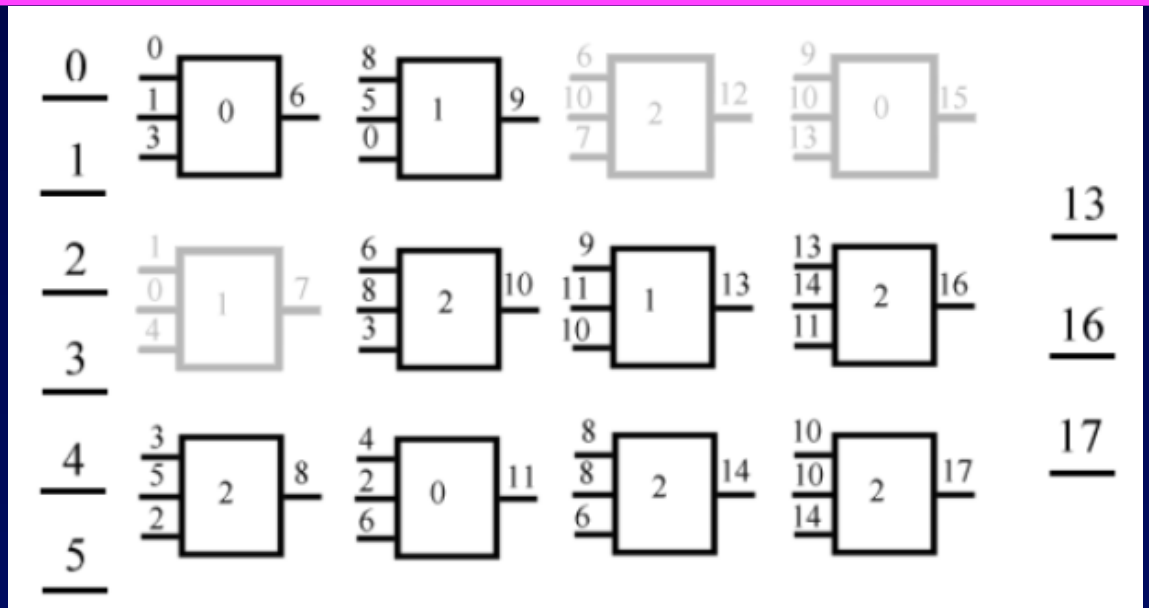


P={G,6,3,3,F,3,3,4,2},
0 1 3 0 – 1 0 4 1 – 3 5 2 2 –
8 5 0 1 – 6 8 3 2 – 4 2 6 0 –
6 10 7 2 – 9 11 10 1 – 8 8 6 2 –
9 10 13 15 – 13 14 11 16 –
10 10 14 2 – 13 16 17

- Un programme cartésien CP est défini par un n-uplet :

$\{G,n_i,n_o,n_n,F,n_f,n_r,n_c,l\}$, avec G : génotype, n_i entrées (inputs), n_o sorties (outputs), n_n arité des opérateurs, F ens. de fonctions, n_f nb de fonctions, n_r nb de lignes (row), n_c nb de colonnes, l nb de colonnes précédentes d'où prendre un paramètre.

Représentation d'un CP



- La fonction est représentée par un chiffre en italiques, et les nœuds « connectés » sont en gras.
- Des nœuds d'une même colonne ne peuvent pas être connectés entre eux.
- Le nombre d'entiers représentant un PC varie entre 0 et $n_r n_c (n_n+1)+n_0$ (longueur variable avec limite supérieure).

Validité des individus

- En PG à la Koza, tous les individus sont valides car écrits en langage fonctionnel (lisp).
- En CGP, il faut vérifier certaines contraintes pour que le résultat d'un opérateur génétique donne toujours un code valide…

Contraintes

- Avec $\{G, n_i, n_o, n_n, F, n_f, n_r, n_c, l\}$, avec G : génotype, n_i entrées (inputs), n_o sorties (outputs), n_n arité des opérateurs, F ens. de fonctions, n_f nb de fonctions, n_r nb de lignes (row), n_c nb de colonnes, l nb de colonnes précédentes d'où prendre un paramètre.
- Lors de la création d'un individu (initialisation, mutation, croisement), il faut que :

Pour c_{kj} la valeur de la k-ième entrée d'un nœud n en colonne j,

$$e_{min} = n_i + (j-l)\, n_r$$
$$e_{max} = n_i + j\, n_r .$$
$$c_{kj} < e_{max}, \; j < l$$
$$e_{min} \le c_{kj} < e_{max}, \; j \ge l$$

Pour c^o_k la valeur de la k-ième valeur de sortie,

$$h_{min} = n_i + (n_c - l)\, n_r$$
$$h_{max} = n_i + (n_c - 1)\, n_r .$$
$$h_{min} \le c^o_k < h_{max}$$

Pour c^f_k la valeur associée à la fonction du k-ième noeud,

$$0 \le c^f_k < n_f$$

Ex : approximation d'un polynome

- Polynome à approximer $x^6 - 2x^4 + x^2$
- 2 Entrées : {1,x}
- Fonctions : {+,-,*,div} (avec div = num si den=0)
- n_i =2 (opérandes), n_o =1 (sorties), n_n=2 (arité), n_f=4 (nb de fonctions), n_r =4 (nb de lignes), n_c =4 (nb de col), l =4 (nb de col précédentes)

1 0 *3*	1 1 *3*	1 1 *2*	0 0 *2*	
4 4 *2*	2 1 *3*	3 4 *1*	1 4 *0*	
6 3 *2*	1 8 *2*	0 8 *2*	8 6 *2*	
6 11 *2*	11 11 *2*	5 3 *3*	13 13 *3*	15