Taxonomia dos Algoritmos Evolutivos
Parte II

1. Programação genética

- É uma tentativa de tratar uma das questões centrais em ciência da computação:

"*Como os computadores podem aprender a resolver problemas sem serem explicitamente programados para tal?*"

- A maior barreira para se atingir este objetivo está na característica determinística da grande maioria dos ramos de pesquisa em inteligência artificial. Nestes casos, as propriedades principais que um programa deve apresentar para ser um candidato à solução de problemas de interesse prático são:

1. Correção;
2. Consistência;
3. Motivação lógica;
4. Precisão;
5. Ordenação;
6. Parcimônia;
7. Definibilidade.

- A programação genética falha em todos estes quesitos. Ela manipula soluções corretas e incorretas, aceita inconsistências e abordagens contraditórias, não apresenta uma variabilidade dinâmica lógica, é predominantemente probabilística, produz soluções não-parcimoniosas e não apresenta um critério de terminação claramente definido.
- Programação genética: *Evolução de programas computacionais usando analogias com muitos dos mecanismos utilizados pela evolução biológica natural*. Pode ser vista como uma extensão dos algoritmos genéticos.
- Efeito prático: Embora haja muito espaço para novas ideias e necessidade de um maior entendimento de todos os mecanismos envolvidos, muitos problemas práticos já têm sido resolvidos com programação genética (PG).
- A PG pode se constituir em uma opção especialmente interessante para o caso de programação de computadores com processamento paralelo.

- Considerando a ideia original da programação genética, um programa computacional vai ser interpretado basicamente como uma sequência de aplicações de funções a argumentos → Paradigma funcional.
- Todos os programas computacionais, independentemente da linguagem em que são implementados, podem ser vistos como uma sequência de aplicações de funções (operações) a argumentos (valores).
- Os compiladores utilizam este fato para, primeiramente, traduzir um dado programa em uma árvore sintática e posteriormente convertê-la em instruções de código assembly e executar o programa.
- A implementação de programação genética é conceitualmente imediata quando associada a linguagens de programação que permitem a manipulação de um programa computacional na forma de uma estrutura de dados (codificação estruturada), inclusive por possibilitar que novos dados do mesmo tipo e recém-criados sejam imediatamente executados como programas computacionais.

- No entanto, qualquer linguagem computacional capaz de implementar (mesmo que indiretamente) a mesma estrutura de dados pode ser potencialmente empregada.
- Linguagem original da programação genética: LISP
- Linguagem mais utilizada nas aplicações recentes: C
- É importante mencionar, entretanto, que, embora a maioria dos autores não utilizem LISP em PG, a maior parte da pesquisa em PG é convenientemente apresentada e discutida utilizando-se uma linguagem do tipo LISP.
- Como qualquer outro sistema computacional inspirado na natureza, a programação genética tem dois propósitos básicos:

1. Servir de ferramenta para a solução de problemas de engenharia;
2. Servir de modelo científico simplificado para processos naturais.

- Na prática, qualquer implementação de programação genética vai envolver, ao menos parcialmente, ambos os propósitos básicos mencionados acima.

- É importante mencionar a existência de várias iniciativas anteriores a KOZA (1992) no sentido de implementar a geração automática de programas computacionais:

1. Programação evolutiva (FOGEL *et al.*, 1966): evolução de programas simples na forma de máquinas de estado finito;
2. Algoritmos genéticos aplicados a tarefas simples de programação automática (CRAMER, 1985).

- John Koza possui uma patente para programação genética, assim como John Holland obteve uma patente para *bucket brigade* (ver Tópico 9).

1 Inspiração biológica

- A biologia evolutiva continua sendo a principal fonte inspiradora da maioria das ideias fundamentais em todas as áreas de computação evolutiva.
- Fontes de entendimento dos processos de computação evolutiva e de inspiração direta para a solução de problemas práticos;
- Fontes de novas ideias para ampliar o poder dos mecanismos evolutivos.

- Veja: EDELSON (2000); JACOB & WEISS (1999); LEWIN (1999); WATSON (1998); WATSON *et al*. (1992).

2 Estruturas de dados

- Computação evolutiva → adaptação evolutiva de estruturas de dados.
- Tipos de dados: primitivos × abstratos ou complexos.
- Estrutura de dados → tipo de dado abstrato.
- Tipos de estruturas de dados mais utilizadas: listas, pilhas, filas, árvores e grafos.
- Estas estruturas são todas organizadas sobre o conceito de nós (algumas vezes chamados de itens ou elementos).
- Um nó é a designação genérica de um dado de um determinado tipo, podendo ser dividido (do ponto de vista lógico) em um ou mais campos de informação, dependendo do nó ser de um tipo primitivo ou abstrato, respectivamente.
- Deste modo, a informação em uma estrutura de dados é organizada na forma de um conjunto de nós, além de haver uma organização interna a cada nó.

- A seleção de qual estrutura de dados é a mais adequada para a representação da informação depende principalmente dos tipos de operações que precisam ser realizadas.

2.1 Lista

- Definição: é um agregado ordenado de elementos que satisfaz as seguintes propriedades:

1. Pode ter zero ou mais elementos;
2. Um novo elemento pode ser inserido em qualquer posição especificada;
3. Qualquer elemento pode ser removido;
4. Pode-se ter acesso a qualquer elemento de uma lista para consulta ou atualização;
5. Pode-se percorrer sequencialmente todos os elementos de uma lista.

- Uma lista linear é um conjunto de nós cujas propriedades estruturais envolvem apenas posições relativas lineares (em uma dimensão) dos nós (ordenação).

2.2 Pilha e fila

- Tanto pilhas como filas são casos particulares de listas.
- Pilhas: são listas lineares em que os elementos só são retirados (*pop*) e inseridos (*push*) numa mesma extremidade que é denominada o topo da pilha. Dizemos então que a manipulação é feita na ordem LIFO (*Last In – First Out*).
- Filas: filas são listas lineares em que os elementos são retirados numa extremidade (término) diferente da extremidade em que eles foram inseridos (começo), na ordem FIFO (*First In – First Out*).

2.3 Árvores

- Definição: árvores são agregados cujos elementos guardam entre si uma relação hierárquica. Formalmente, uma árvore é um conjunto finito *T* de nós, tais que:

1. Existe um nó que não é subordinado a nenhum outro nó, denominado raiz da árvore.

2. Os demais nós (ligados à raiz por arcos) formam *m* conjuntos disjuntos *S*1,*S*2,...,*Sm* , onde cada um destes conjuntos é uma (sub-)árvore.
3. os nós subordinados a um nó e ligados a ele por um arco são denominados os filhos daquele nó, que por sua vez é denominado de pai. Dizemos também que esses nós subordinados são irmãos entre si. O número de filhos de um nó é o grau daquele nó. Um nó de grau zero é chamado folha ou nó terminal. O nível de um nó é a distância daquele nó até a raiz (número de arcos que devem ser percorridos). A altura de uma árvore é o maior nível das folhas daquela árvore.

- As árvores mais utilizadas em computação são as denominadas árvores ordenadas, nas quais uma alteração na ordem das (sub-)árvores ligadas a cada nó pode provocar interpretações erradas da relação hierárquica existente na árvore.
- Quando se limita o grau máximo de uma árvore a *n*, então a árvore é chamada de árvore *n*-ária. Dentre as árvores ordenadas de grau limitado destaca-se a árvore binária, onde cada nó tem zero (folhas) ou dois filhos (nós internos).

- As operações básicas para manipulação de uma árvore genérica (para qualquer dado armazenado em cada nó) são:
  1. Inicialização;
  2. Criação de uma árvore;
  3. Remoção de uma árvore;
  4. Caminhada por uma árvore.
- Observe que a inserção e a remoção de um nó de uma árvore não são consideradas operações básicas. A razão disso é que essas operações dependem do dado armazenado no nó manipulado, podendo até modificar a relação hierárquica entre os nós da árvore.
- A caminhada por uma árvore é a principal operação básica. Através dela pode-se percorrer todos os nós de uma árvore sequencialmente e ter acesso a um nó específico. Muitas vezes, a correção dos dados lidos de uma árvore depende essencialmente da forma como ela foi percorrida.

2.4 Grafos

- Definição: constituem um tipo de agregado capaz de representar relações binárias entre os seus elementos.
- Sendo as relações entre os elementos assimétricas, teremos um grafo dirigido (*directed graph*). Senão, dizemos que o grafo é não-dirigido (*undirected graph*).
- Matematicamente, um grafo $G(N,A)$ é constituído por um conjunto de N vértices ou nós (elementos) e A arcos ou arestas (relações binárias). Os vértices podem ser referenciados por seus rótulos (normalmente, números inteiros) e os arcos, pelos vértices que conectam.
- Um caminho é uma sequência de arcos em que o segundo nó de cada arco coincide com o primeiro nó do arco seguinte e cada nó pertence, no máximo, a dois arcos. Quando o primeiro nó coincide com o último nó de um caminho, há um circuito.
- Uma árvore é, portanto, um grafo para o qual existe um, e somente um, caminho entre qualquer par de nós, pois não se estabelecem circuitos neste caso.

- Um grafo não-conexo (*unconnected graph*) tem pares de nós não ligados pelos arcos. Nós ligados por arcos são chamados adjacentes.
- Grafos ponderados representam extensões em que os arcos passam a ter pesos e as relações entre os nós deixam de ser binárias.

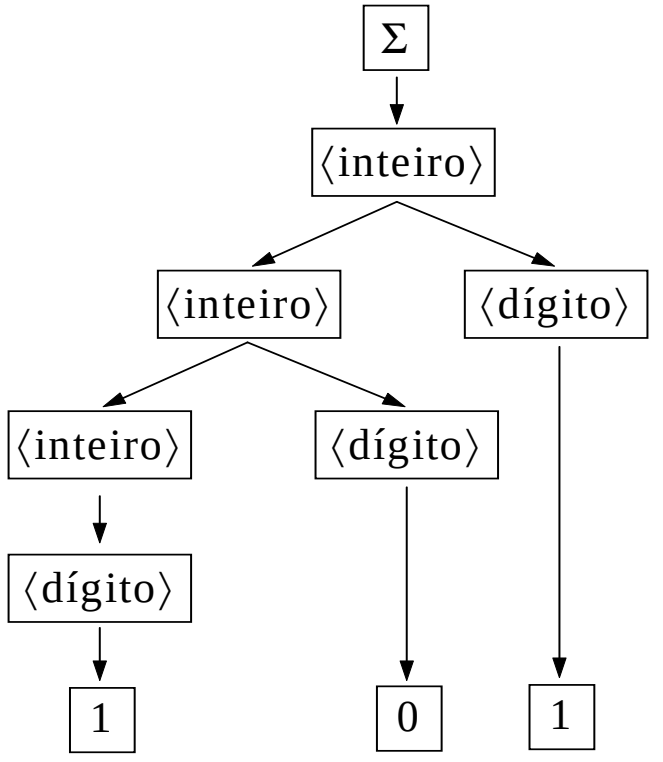
3 Árvores sintáticas (*Parse Trees*)

- Considere uma gramática $G = \{N, T, P\}$. Dada uma sentença representando uma concatenação arbitrária de símbolos terminais de G, é possível verificar se esta sentença pertence ou não a G pela construção (ascendente ou descendente) de uma árvore sintática que tenha o símbolo não-terminal inicial Σ como raiz e símbolos terminais como folhas (nós de grau zero).

◊ $N = \{\Sigma, \langle\text{inteiro}\rangle, \langle\text{dígito}\rangle\}$

◊ $T = \{1,0\}$

◊ $P = \{$
$\Sigma \rightarrow \langle\text{inteiro}\rangle;$
$\langle\text{inteiro}\rangle \rightarrow \langle\text{inteiro}\rangle\ \langle\text{dígito}\rangle;$
$\langle\text{inteiro}\rangle \rightarrow \langle\text{dígito}\rangle;$
$\langle\text{dígito}\rangle \rightarrow 1;$
$\langle\text{dígito}\rangle \rightarrow 0$
$\}$

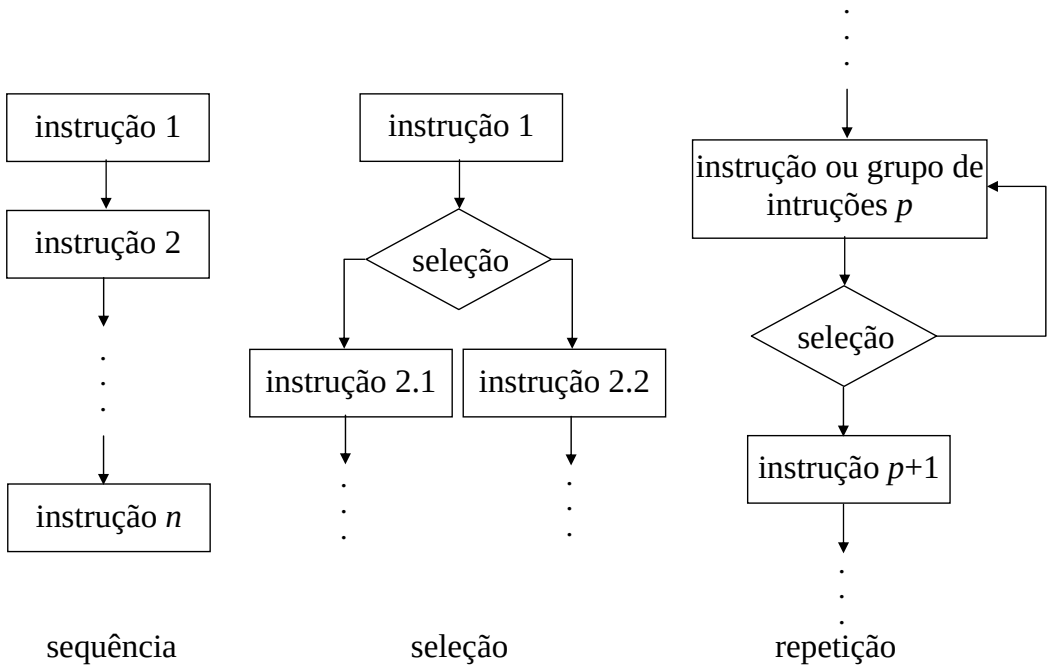


4 Introdução à linguagem LISP (*LISt Processing*)

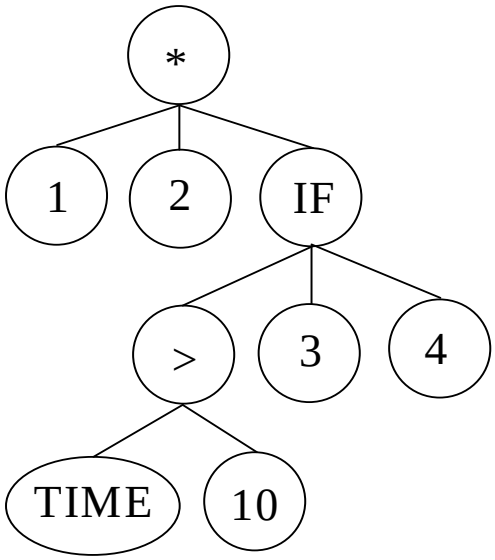
- Princípios da linguagem LISP serão apresentados a seguir, porque esta linguagem foi utilizada originalmente para a implementação da programação genética.
- No entanto, qualquer linguagem computacional suficientemente complexa é passível de utilização para a implementação de programação genética.
- Em LISP, existem apenas dois tipos de entidades: átomos (constantes e variáveis) e listas (conjunto ordenado de itens delimitados por um par de parênteses).
- Expressão simbólica (*S-expression*): é um átomo ou uma lista. É a única forma sintática existente, ou seja, todos os programas em LISP são expressões simbólicas.
- Avaliação de uma lista: o primeiro elemento da lista é tomado como uma função a ser aplicada aos outros elementos da lista. Isto implica que os demais elementos da lista devem ser previamente avaliados antes de serem utilizados como argumentos da função, representada pelo primeiro elemento da lista (notação com prefixação).

- Exemplo 1: **`(+ 7 2)`** é uma expressão simbólica em LISP. Esta expressão simbólica solicita a aplicação da função **`+`** a dois argumentos representados pelos átomos **`7`** e **`2`**. O valor retornado como o resultado da avaliação desta expressão simbólica é **`9`**.
- Exemplo 2: **`(+ (* 2 3) 4)`** é uma expressão simbólica em LISP em que um dos argumentos também é uma lista. O valor retornado como o resultado da avaliação desta expressão simbólica é **`10`**.
- Conclusão: Os programas computacionais em LISP podem ser vistos como uma composição de funções.
- No entanto, estas funções não se restringem a operadores aritméticos simples. É possível implementar todos os tipos de construção admitidas para um programa computacional (sequência, seleção e repetição) com base na notação com prefixação do LISP.

- Fluxo de controle: sequência, seleção e repetição



- *A adequação da linguagem LISP para programação genética está no fato de que as expressões simbólicas representam diretamente a árvore sintática de um programa.*
- Exemplo 3: **`(* 1 2 (IF (> TIME 10) 3 4) )`**



5 Componentes de representação para PG

- Para a programação genética, um problema é definido pela representação e pela função de *fitness*.
- Para se obter uma representação expressiva e uma função de *fitness* efetiva, é necessário ser um conhecedor profundo do problema que se está tentando resolver, inclusive tendo a capacidade de abordar o problema de maneira não-usual. Raramente o conhecimento prévio de um dado problema é suficiente para obter sua solução usando programação genética.

5.1 Representação

- A representação consiste de funções e terminais, definindo uma linguagem.
- Os programas computacionais na linguagem definida são os indivíduos, passíveis de representação em estruturas de dados do tipo árvore.

- Estes programas precisam ser executados para se obter o correspondente fenótipo do indivíduo.
- Aspectos a serem considerados na definição da representação de um problema:

1. *Propriedade de fechamento* (condições de contorno para as funções): é necessário examinar os valores de retorno de todas as funções e terminais, e estar certo de que todas as funções vão aceitar como argumento qualquer valor e tipo de dado que possa ser retornado por uma função do conjunto de funções, ou então por um terminal do conjunto de terminais. É possível definir também rotinas de tratamento especial para exceções pontuais.
2. *Propriedade de suficiência:* A definição das funções e terminais acaba especificando o espaço de busca dos possíveis programas. Obviamente, este espaço deve ser suficientemente amplo para conter a solução desejada. Além disso, as funções e terminais definidos devem ser apropriados para o domínio do problema. No entanto, quanto maior for este espaço, menor é a chance de

encontrar a solução. Mesmo com o conhecimento preciso do domínio do problema, fica definido um compromisso entre potencial de representação e complexidade do processo de busca.

3. A impossibilidade de manter soluções sintaticamente incorretas e/ou semanticamente inválidas pode produzir problemas de baixa diversidade populacional.

5.1.1 Tipo de funções na representação

- Funções aritméticas padrões;
- Funções de programação padrões;
- Funções matemáticas padrões;
- Funções lógicas;
- Funções de domínio específico.

5.1.2 Tipos de terminais na representação

- Os terminais correspondem às entradas dos programas a serem evoluídos. Eles recebem este nome por corresponderem às folhas (nós de grau zero) das estruturas em árvore que representam os programas computacionais.
- Podem ser variáveis, constantes ou funções que não recebem argumentos.

5.2 Função de *fitness*

- A função de *fitness* é excepcionalmente importante, por ser a única fonte de representação do que se intenciona resolver (otimizar, aproximar, modelar,etc.).
- A função de *fitness* deve medir não apenas a adaptabilidade do indivíduo (programa computacional), mas relacionar a adaptabilidade de um indivíduo com a de outros, além de ser capaz de distinguir uma solução mais completa de uma menos completa (*partial credit*).

- Geralmente, cada indivíduo da população é avaliado em várias situações diferentes e com mais de uma função de *fitness* (teste de software), e seu nível de adaptação é extraído da soma ponderada dos níveis de adaptação obtidos para cada situação.
- Alternativa: técnicas de programação multiobjetivo (ver Tópico 10).

6 O que distingue programação genética de algoritmos genéticos?

- A programação genética é uma adaptação dos algoritmos genéticos, mas de algum modo ela está mais para uma generalização do que para uma especialização da sua disciplina-pai.
- Ausência de codificação prévia da solução do problema;
- Cada indivíduo da população (candidato à solução) é um programa computacional. A solução (ou parte dela) é obtida pela execução do programa e não por uma

codificação direta, como em algoritmos genéticos (a execução do programa não deve ser entendida apenas em seu sentido estrito).

- Material genético estruturado hierarquicamente (estrutura em árvore), e não apenas ordenadamente;
- Material genético de tamanho variável durante o processo evolutivo (necessidade do emprego de mecanismos de limitação de tamanho);
- *Crossover* com preservação da sintaxe correta do material genético.
- A variabilidade de comportamento do processo evolutivo em programação genética é geralmente maior que em algoritmos genéticos, ou seja, muitas "rodadas" devem ser executadas para se extrair algum tipo de tendência em relação aos resultados.
- Passos preparatórios para aplicação de PG e GA envolvem a determinação de:

Algoritmos genéticos	**Programação genética**
representação (codificação)	conjunto de terminais
	conjunto de funções primitivas
função de *fitness*	função de *fitness*
parâmetros para controlar a execução	parâmetros para controlar a execução
método de designação do resultado e do critério de terminação	método de designação do resultado e do critério de terminação

- Passos iterativos da programação genética (poucas diferenças de concepção em relação aos algoritmos genéticos):

1. Gerar uma população inicial de composições aleatórias de funções e terminais (programas computacionais);
2. Executar iterativamente os seguintes passos, até que o critério de parada tenha sido satisfeito:

(a) Execute cada programa da população e atribua um valor de adaptação usando a função de *fitness*;

(b) Crie uma nova população (geração) pela aplicação dos operadores de recombinação, possivelmente mutação (ausente nas versões originais de algoritmos para PG), e seleção junto à população atual, baseado nos valores de adaptação associados a cada indivíduo;

(c) Avalie o resultado obtido nesta nova geração e teste o critério de parada.

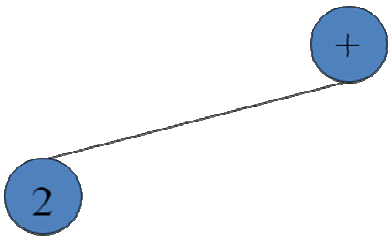
7 Inicialização da população

- Inicialmente deve-se definir os conjuntos de símbolos terminais e não-terminais que poderão compor o programa.
- Em seguida, escolhe-se aleatoriamente um elemento de algum desses conjuntos para iniciar a construção da árvore. Para evitar árvores muito curtas, pode-se forçar, inicialmente, a escolha de elementos do conjunto de símbolos não-terminais, ou seja, a escolha de funções. Uma vez escolhida uma função, é necessário definir os seus argumentos, que podem vir a ser sub-árvores.

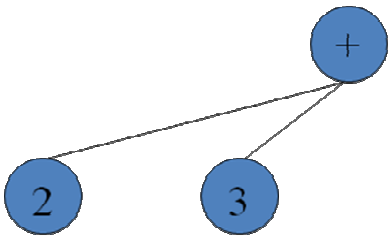
- Exemplo 4:
  - Conjunto de símbolos não-terminais (funções) = {+, –, *, /}
  - Conjunto de símbolos terminais = {inteiros, X, Y}

(+ …)

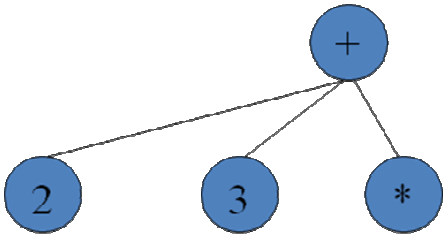
(+ 2 …)



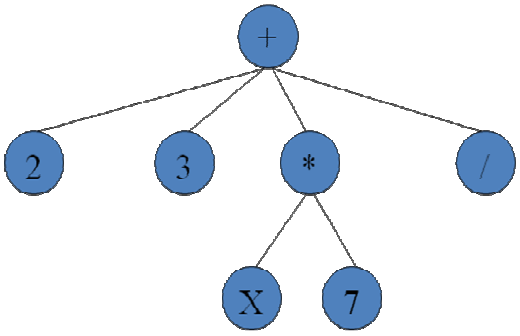
(+ 2 3 …)



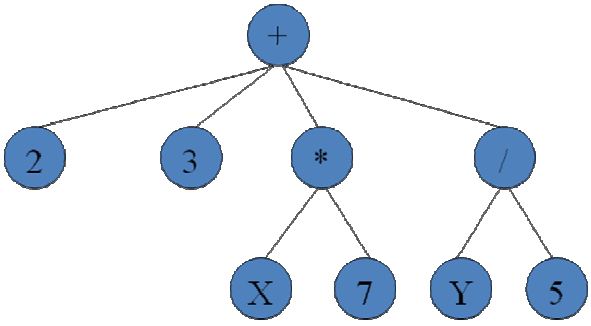
(+ 2 3 (* …) …)



(+ 2 3 (* X 7) (/ …))



(+ 2 3 (* X 7) (/ Y 5))



8 Recombinação e mutação em programação genética

- O operador de recombinação ou *crossover* é utilizado para gerar indivíduos (programas computacionais) para a próxima geração a partir de dois programas-pais escolhidos com base no valor fornecido pela função de *fitness* para cada indivíduo da geração atual. Os programas-pais são geralmente de tamanho e forma diferentes.
- Os programas-filhos são compostos de sub-árvores dos programas-pais, podendo ser de tamanho e forma diferentes daqueles apresentados pelos programas-pais.
- Intuitivamente, se dois programas-pais são de algum modo eficientes na solução de um dado problema, então pelo menos algumas de suas sub-árvores constituintes têm algum mérito. Sendo assim, a recombinação aleatória pode acabar criando um programa-filho mais apropriado para resolver o problema que seus programas-pais.

- Embora não seja adotada em boa parte das implementações de programação genética, a mutação pode ser incorporada, desde que leve em consideração o caráter hierárquico dos programas computacionais.
- A mutação pode ser muito útil na manutenção da diversidade populacional e na variabilidade dinâmica dos programas em evolução.
- **Na ausência de mutação, a população inicial deve ser suficientemente numerosa e diversificada no sentido de conter uma diversidade de programas cujos blocos constituintes (sub-árvores) possam ser adequadamente reagrupados, apenas pelo operador de *crossover*, para produzir uma solução para o problema.**

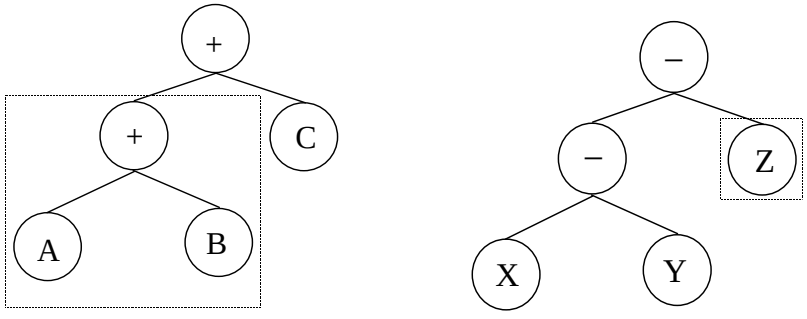
8.1 Operador de recombinação (*crossover*) e expressões simbólicas

- Como em algoritmos genéticos, a aplicação do operador de *crossover* requer como entrada duas expressões simbólicas, denominadas expressões-pai. No entanto, aqui a recombinação não pode ser aleatória, devendo respeitar a sintaxe da construção.

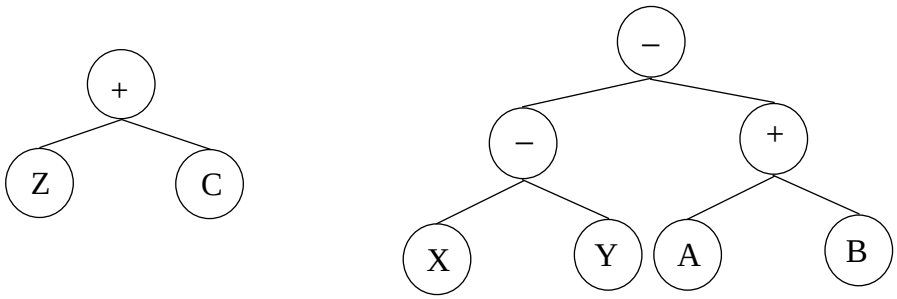
- Isto implica, por exemplo, a definição de pontos de *crossover* independentes para cada uma das expressões-pai.
- O ponto de *crossover* isola uma sub-árvore de cada expressão-pai, cada qual podendo ter alturas arbitrárias (dentro do limite estabelecido para a aplicação).
- Exemplo 5: Considere as expressões-pai a seguir:

```
(+ (+ A B) C)          (- (- X Y) Z)
```

com as respectivas árvores sintáticas dadas na forma:



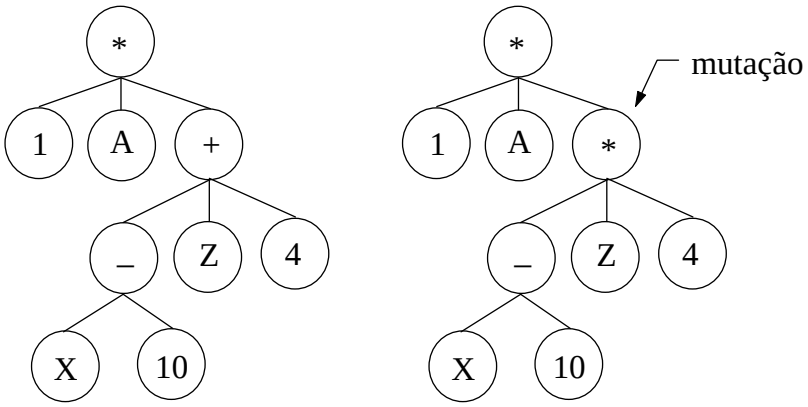
- Supondo que os pontos de *crossover* isolaram as sub-árvores contidas nos retângulos tracejados, as expressões-filho resultantes são dadas a seguir:



8.2 Operador de mutação e expressões simbólicas

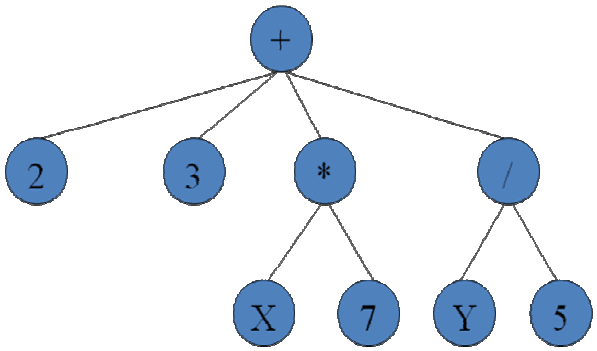
- A mutação pode ser útil, por exemplo, na introdução de modificações nas raízes das sub-árvores existentes, mantendo inalterados todos os ramos que dali partem.

- Exemplo 6:

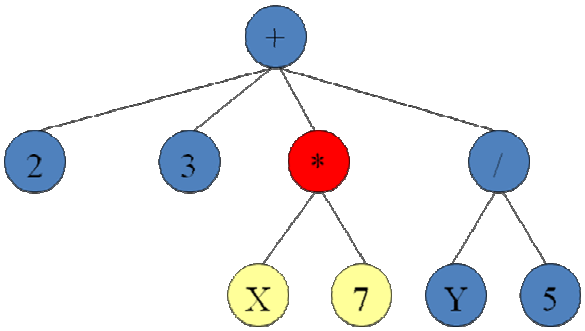


Exemplo 7:

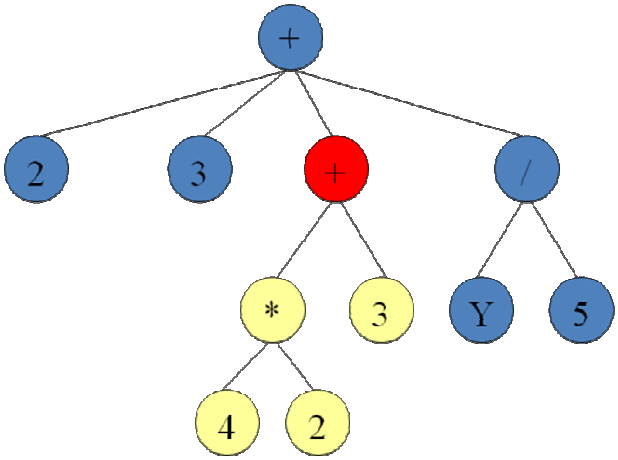
(+ 2 3 (* X 7) (/ Y 5))



- Inicialmente tome um nó aleatório: (+ 2 3 **(* X 7)** (/ Y 5))



- Em seguida, o substitua por uma subárvore gerada aleatoriamente, como feito para a população inicial: (+ 2 3 **(+ (* 4 2) 3)** (/ Y 5))



9 Questões em aberto

- O que faz um problema ser fácil ou difícil de resolver usando PG?

- Sob o ponto de vista da programação genética, quão mais complexos vão ficando os problemas à medida que sua solução exige a obtenção de programas maiores?
- Como fica o desempenho da programação genética à medida que se diminui ou aumenta o número de elementos do conjunto de funções e/ou terminais?
- Que tamanho de população é mais adequado para resolver cada tipo de problema?
- Deve-se usar mutação ou enriquecer a população inicial?
- Sabendo-se que o conjunto de casos passíveis de incorporação no processo de definição da adaptabilidade de um programa geralmente é muito maior que aquele utilizado na prática, quão bem os programas resultantes do processo evolutivo generalizam quando submetidos a casos não incorporados no processo de definição da função de *fitness*?
- Até que ponto um programa pode ser otimizado ao mesmo tempo quanto a correção, tamanho e eficiência?

10 Extensões e abordagens alternativas

1. Introdução de primitivas de ordem superior: edição, encapsulamento e definição automática de funções.
2. Comparação sistemática com outros métodos utilizados em ciência da computação para busca em árvores, particularmente em árvores sintáticas.
3. Comparação com outras técnicas de indução de programas e aprendizado de máquina.
4. Emprego de arquiteturas híbridas, incluindo componentes de programação genética.
5. Emprego da teoria de autômatos celulares e algoritmos genéticos na computação automática (emergência) de soluções em sistemas complexos (CRUTCHFIELD & MITCHELL, 1995; MITCHELL *et al.*, 1994; MITCHELL *et al.*, 1993).
6. Linear genetic programming (BRAMEIER & BANZHAF, 2007).

11 Atributos para programação automática

- Programação automática (WYWIWYG - *What You Want Is What You Get*)
- Dez atributos para programação automática (KOZA *et al.*, 1998):

1. Iniciar a partir de uma definição de alto nível acerca dos requisitos para se chegar à solução de um problema;
2. Produzir uma "entidade" que seja executável em um computador;
3. Definir automaticamente o tamanho e a forma da solução: número, tipo e sequência de passos primitivos para se chegar à solução;
4. Realizar encapsulamento: definir grupos de passos primitivos e reutilizá-los com diferentes instâncias;
5. Organizar estes grupos de passos em uma estrutura hierárquica;
6. Implementar todos os tipos de construção admitidas para um programa computacional: sequência, seleção e repetição;

7. Ser independente do problema no sentido de não necessitar modificar os passos executáveis do sistema de programação genética para cada novo problema;
8. Produzir uma solução satisfatória para uma grande variedade de problemas nas mais diversas áreas de atuação científica;
9. Apresentar uma degradação de desempenho suave para versões maiores de um mesmo problema;
10. Produzir soluções competitivas com aquelas fornecidas pelos especialistas nas respectivas áreas de aplicação.

- A programação genética é mais que apenas um mecanismo de programação automática. Independente do genótipo ser interpretado como um programa, no campo dos algoritmos evolutivos, a programação genética abriu as portas para:

1. Genomas de comprimento variável;

2. Incorporação em algoritmos evolutivos do conceito de que pode-se criar representação estruturada pela seleção natural e pelos efeitos criativos da recombinação e da mutação;
3. Uma variedade de novos operadores genéticos.

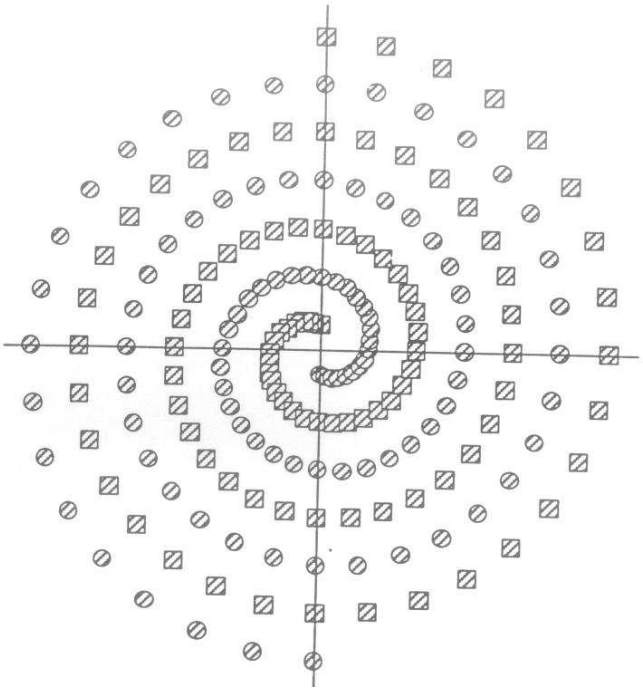
- A programação genética pode ser aplicada a outros problemas que podem ser expressos em estruturas de dados hierárquicas. Ex: Definição de arquiteturas de redes neurais artificiais (GRUAU, 1994).
- Outros exemplos de aplicação: desenvolvimento de geradores de números aleatórios, reconhecimento de objetos 3D, desenvolvimento de classificadores, processamento de linguagem natural, geração automática de planos para robôs móveis, projeto de circuitos integrados, composição musical, geração de imagens, regressão simbólica, reconhecimento de padrões.

12 Exemplos de programas

- Programa evoluído para representar uma função lógica que atendesse uma certa tabela-verdade:

```
(NOT (AND (OR (AND (OR (OR (AND (OR (OR (OR S (OR (OR W
(NOT E)) N)) N) N) (AND (OR (OR (OR (OR (AND (OR W N)
(NOT E)) (NOT (NOT X))) (AND S X)) (NOT (NOT X))) (AND S
X)) (NOT (NOT (AND (OR W N) (NOT E)))))) (NOT (NOT X)))
(AND S X)) (NOT (NOT N))) (NOT (OR (OR X N) (NOT E))))
(OR (AND (NOT (OR (OR W (NOT (NOT X))) N)) (OR (OR (AND
(OR (OR W (NOT (NOT X))) N) (AND S X)) (AND (AND S S) (OR
(OR (AND (OR W N) (NOT E)) (NOT (NOT X))) (AND S X))))
(NOT (OR (AND (AND (OR (AND S X) N) (AND S X)) (NOT E))
(AND (AND S S) (OR (OR W (OR W W)) N)))))) (OR (OR W (OR
W W)) N)))).
```

- Problema de classificação de padrões: 2 espirais com 194 pontos; cada espiral representa uma classe.




```
(SIN (IFLTE (IFLTE (+ Y Y) (+ X Y) (- X Y) (+ Y Y)) (* X
X) (SIN (IFLTE (% Y Y) (% (SIN (SIN (% Y 0.30400002))) X)
(% Y 0.30400002) (IFLTE (IFLTE (% (SIN (% (% Y (+ X Y))
0.30400002)) (+ X Y)) (% X -0.10399997) (- X Y) (* (+
-0.12499994 -0.15999997) (- X Y))) 0.30400002 (SIN (SIN
(IFLTE (% (SIN (% (% Y 0.30400002) 0.30400002)) (+ X Y))
(% (SIN Y) Y) (SIN (SIN (SIN (% (SIN X) (+ -0.12499994
-0.15999997))))) (% (+ (+ X Y) (+ Y Y)) 0.30400002)))) (+
(+ X Y) (+ Y Y))))) (SIN (IFLTE (IFLTE Y (+ X Y) (- X Y)
(+ Y Y)) (* X X) (SIN (IFLTE (% Y Y) (% (SIN (SIN (% Y
0.30400002))) X) (% Y 0.30400002) (SIN (SIN (IFLTE (IFLTE
(SIN (% (SIN X) (+ -0.12499994 -0.15999997))) (% X
-0.10399997) (- X Y) (+ X Y)) (SIN (% (SIN X) (+
-0.12499994 -0.15999997))) (SIN (SIN (% (SIN X) (+
-0.12499994 -0.15999997)))) (+ (+ X Y) (+ Y Y))))))) (% Y
0.30400002))))).
```

13 Referências

ANGELINE, P.J., KINNEAR JR., K.E. (eds.) "Advances in Genetic Programming". volume II, MIT Press, 1996.

BANZHAF, W., NORDIN, P., KELLER, R.E. & FRANCONE, F.D. "Genetic Programming – An Introduction : On the Automatic Evolution of Computer Programs and Its Applications". Morgan Kaufmann Publishers, 1998.

BOJARCZUK, C.C. "Programação Genética com Restrições para Descoberta de Conhecimento em Bases de Dados Médicos". Dissertação de Mestrado, Centro Federal de Educação Tecnológica do Paraná (CEFET-PR), 2001.

BRAMEIER, M. & BANZHAF, W. "Linear Genetic Programming", Springer-Verlag, 2007.

CRAMER, N.L. "A representation for the adaptive generation of simple sequential programs". *in* J.J. Grefenstette (ed.) *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, 1985.

CRUTCHFIELD, J.P. & MITCHELL, M. "The evolution of emergent computation". *Proceedings of the National Academy of Science*, USA, vol. 92, pp. 10742-10746, 1995.

EDELSON, E. "Francis Crick and James Watson : And the Building Blocks of Life". Oxford Portraits in Science, Oxford University Press, 2000.

FOGEL, L.J., OWENS, A.J. & WALSH, M.J. "Artificial Intelligence through Simulated Evolution". John Wiley, 1966.

GRUAU, F. "Genetic Micro Programming of Neural Networks". *in* Kinnear Jr., K.E. (ed.) Advances in Genetic Programming, Mit Press, chapter 24, pp. 495-518, 1994.

JACOB, F & WEISS, G. "Of Flies, Mice, and Men". Harvard University Press, 1999.

KINNEAR JR., K.E. (ed.) "Advances in Genetic Programming". MIT Press, 1994.

KOZA, J.R. "Genetic Programming: On the Programming of Computers by means of Natural Selection", MIT Press, 1992.

KOZA, J.R. "Genetic Programming II: Automatic Discovery of Reusable Programs", MIT Press, 1994.

KOZA, J.,R. BENNETT III, F.H., ANDRE, D. & KEANE, M.A. "Genetic Programming III: Darwinian Invention and Problem Solving". Morgan Kaufmann Publishers, 1999.

LEWIN, B. "Genes VII". Oxford University Press, 1999.

MITCHELL, M., CRUTCHFIELD, J.P. & HRABER, P.T. "Evolving cellular automata to perform computations: Mechanisms and impediments". *Physica D*, vol. 75, pp. 361-391, 1994.

MITCHELL, M., HRABER, P.T. & CRUTCHFIELD, J.P. "Revisiting the edge of chaos: Evolving cellular automata to perform computations". *Complex Systems*, vol. 7, pp. 89-130, 1993.

SPECTOR, L., LANGDON, W.B., O' REILLY, U.-M., ANGELINE, P.J. (eds.) "Advances in Genetic Programming". volume III, MIT Press, 1999.

WATSON, J.D "The Double Helix : A Personal Account of the Discovery of the Structure of DNA". Simon & Schuster, Reprint Edition, 1998.

WATSON, J.D., GILMAN, M., WITKOWSKI, J., ZOLLER, M. & WITKOWSKI, G. "Recombinant DNA". W H Freeman & Co., 2nd. edition, 1992.