OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

INF FAKULTÄT FÜR INFORMATIK

Evolutionary Algorithms

Genetic Programming

Prof. Dr. Rudolf Kruse **Pascal Held**

{kruse,pheld}@iws.cs.uni-magdeburg.de

Otto-von-Guericke University Magdeburg

Faculty of Computer Science

Institute of Knowledge and Language Engineering

Outline

Genetic Programming

Genetic programming (GP) is based on the following ideas:

- describing a solution for a problem by some computer program that is connecting a certain input with certain output
- searching for a matching computer program
- universal way of learning computer programs
- representating programs by parse trees

Learning Programs

Many problems can be seen as „learned programs“, e.g.:

- controlling
- designing
- searching
- representing knowledge
- symbolic regression
- induction of decision trees

Genetic Programming

representation of candidate solutions

- **previously:** by chromosomes of fixed length (vector of genes)
- **now:** by function expressions or programs, i.e.
  - complex chromosomes of variable length

technical fundamentals: grammar for describing a language

- choose two sets:

  $\mathcal{F}$ – set of function symbols and operators
  $\mathcal{T}$ – set of terminal symbols (constants and variables)

These sets $\mathcal{F}$ and $\mathcal{T}$ are specific for each problem. They shouldn't be too large (thus limiting the search space) but large enough to allow for finding a solution

Examples for Symbol Sets

- **example 1:** learning a boolean function
  - $\mathcal{F} = \{\text{and}, \text{or}, \text{not}, \text{if} \ldots \text{then} \ldots \text{else} \ldots, \ldots\}$
  - $\mathcal{T} = \{x_1, \ldots, x_m, 1, 0\}$ bzw. $\mathcal{T} = \{x_1, \ldots, x_m, t, f\}$

- **examle 2:** symbolical regression
  - regression: finding an approximating function for given data while minimizing the sum of squared errors
    $\rightarrow$ *method of least squares*
  - $\mathcal{F} = \{+, -, *, /, \sqrt{}, \sin, \cos, \log, \exp, \ldots\}$
  - $\mathcal{T} = \{x_1, \ldots, x_m\} \cup \mathbb{R}$

Closure of $\mathcal{F}$ and $\mathcal{T}$

$\mathcal{F}$ and $\mathcal{T}$ should be closed so there are no program crashes if incorrect parameter(types) are passed to function symbols.

different strategies can guarantee closure, e.g.

- implementing stable operators instead of instable ones, e.g.
  - safe division, giving 0 or a maximum value
  - safe root, operating on absolute values
  - safe logarithm: $\forall x \leq 0 : \quad \log(x) = 0$ or similar
- combination of several different function types
  - e.g. numerical and boolean values $(\text{FALSE} = 0, \text{TRUE} \neq 0)$
- implementation of conditional comparators
  - e.g. *IF* $x < 0$ *THEN* ...
- ...

Completeness of $\mathcal{F}$ and $\mathcal{T}$

A GP can only solve problems efficiently and effective, if function- and terminal symbol sets are sufficient/complete to ensure an appropriate program can be found.

In boolean algebra $\mathcal{F} = \{\wedge, \neg\}$ and $\mathcal{F} = \{\rightarrow, \neg\}$ are complete operator sets, $\mathcal{F} = \{\wedge\}$ is not.

- general problem of machine learning: **feature selection**
- finding the smallest sufficient set is (often) NP-hard
- often there are more functions within $\mathcal{F}$ than necessary

Symbolic Expressions

chromosomes = expressions (composition of elements from $\mathcal{C} = \mathcal{F} \cup \mathcal{T}$ and maybe brackets)

restriction to "well-formed" expressions

recursive definition (prefix notation):

- Symbols for constants and variables are symbolic expressions.
- If $t_1, \ldots, t_n$ are symbolic expressions, and $f \in \mathcal{F}$ is an (n-ary) operator symbol, then $(f t_1 \ldots t_n)$ is a symbolic expression, too.
- No other string is called symbolic expression.

examples:

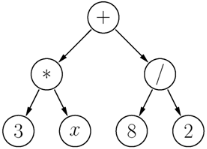
- „$(+\ (*\ 3\ x)\ (/\ 8\ 2))$" is a symbolic expression
  Lisp- or Scheme-like notation, means: $3 \cdot x + \frac{8}{2}$
- „$2\ 7\ *\ (3\ /$" is not a symbolic expression

Implementation

- implementation of GPs: represent symbolic expressions by so-called parse trees
  (parse trees are used by parsers to represent and optimize arithmetical expressions)

symbolic expression:
$(+ \ (* \ 3 \ x) \ (/ \ 8 \ 2))$

parse tree:



Within Lisp/Scheme expressions are nested lists:
first list element is a function symbol or operator,
following elements are arguments or operators.

Processing Genetic Programs

- generate an **initialization population** of random expressions
- **evaluate** these expressions by calculating their fitness

  learning boolean functions: ratio of correct output for all inputs given to a sample

  symbolic regression: sum of squared errors of the given measurement points

  1-D: data (x_i, y_i), $i = 1, \ldots, n$, fitness $f(c) = \sum_{i=1}^{n}(c(x_i) - y_i)^2$
- **selection** with one of the discussed procedures
- application of **genetic operators**, usually only crossover

Outline

1. Motivation

2. **Initialization**
   - „grow“
   - „full“
   - „ramped half-and-half“

3. Genetic Operators

4. Examples

5. Summary and Prospect

Initialization of a GP-Population

parameter for the process of creation:

- maximal nesting (maximal tree height) d_{max} or
- maximal number of tree nodes n_{max}

three different ways for initialization [Koza, 1992]:

1. *grow*
2. *full*
3. *ramped half-and-half*

- as with EAs duplicates can be avoided
- call to *grow* and *full*: initialize(root, 0)

„grow“

Algorithm 1 initialize-grow

Input: node n, depth d, maximumDepth $d_{\max}$

if $d = 0$ {
  $n \leftarrow$ draw a node from $\mathcal{F}$ uniformly distributed
} **else** { **if** $d = d_{\max}$ {
  $n \leftarrow$ draw a node from $\mathcal{T}$ uniformly distributed
} **else** {
  $n \leftarrow$ draw a node from $\mathcal{F} \cup \mathcal{T}$ uniformly distributed
}
if $n \in \mathcal{F}$ {
  for each $c \in$ arguments of n {
    initialize-grow($c, d + 1, d_{\max}$)
  }
} **else** {
  return
}

- generates trees of irregular shape
- nodes: randomly drawn from $\mathcal{F}$ *and* $\mathcal{T}$ (except root)
- branch with terminal symbol ends prior to reaching maximum

„full“

Algorithm 2 initialize-full

Input: nodes n, depth d, maximum depth $d_{\max}$

```
if d ≤ d_max {
    n ← draw nodes from 𝓕 uniformly distributed
    for each c ∈ arguments of n {
        initialize-full(c, d + 1, d_max)
    }
} else {
    n ← draw nodes from 𝓣 uniformly distributed
}
return
```

generates balanced trees

nodes: randomly drawn *only* from $\mathcal{F}$ (except at maximum depth)

at maximum depth: randomly draw *only* from $\mathcal{T}$

„ramped-half-and-half“

Algorithm 3 initialize-ramped half-and-half

Input: maximum depth $d_{\max}$, population size μ (even multiple of $d_{\max}$)

```
P ← ∅
for i ← 1 ... d_max {
    for j ← 1 ... μ/(2 · d_max) {
        P ← P ∪ initialize-full(root, 0, i)
        P ← P ∪ initialize-grow(root, 0, i)
    }
}
```

combination of *grow* and *full* methods

- generates even number of grown and full trees with all possible depths between 1 and $d_{\max}$

large variation of trees and shapes

- suitable for GP (see evolutionary principles)

Outline

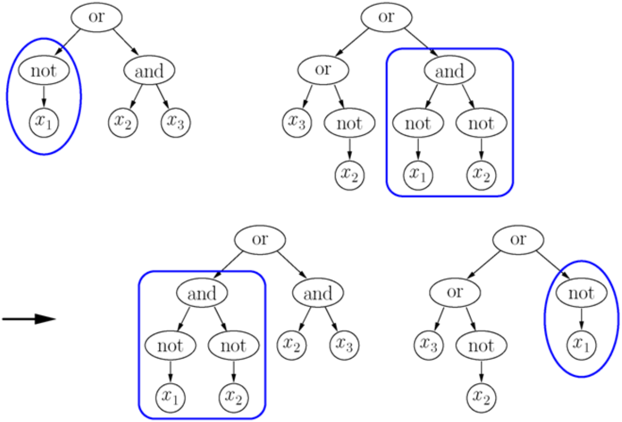
Genetic Operators

- usually initialized population has no good fitness
- the evolutionary progress changes population via genetic operators
- for GPs: many different genetic operators
- the 3 most important operators are:

  crossover,
  mutation and
  clonale reproduction (duplication of an individual)

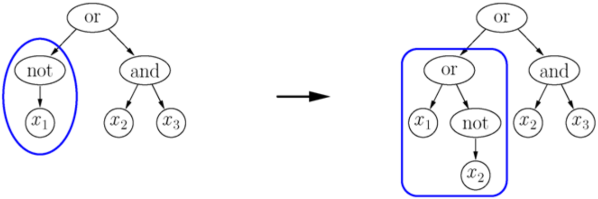
Crossover

- exchanging two subexpressions (subtrees)



Mutation

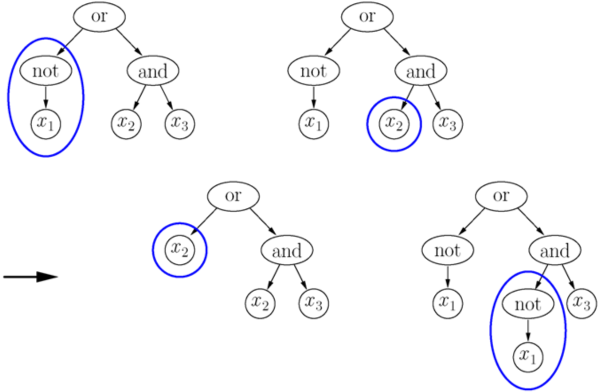
exchanging a subexpression (subtree) by randomly generated one:



- if possible, exchange only small subtrees
- with very large population: sufficiently large stock of “genetic material“, so often only crossover is used and no mutation

Advantage of Crossover

crossover is more powerful with GPs than with vectors:
crossover of identical parental programs might create different individuals



Outline

1. Motivation

2. Initialization

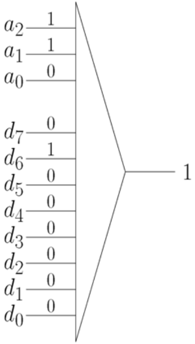
3. Genetic Operators

4. **Examples**
   - 11-Multiplexer
   - Stimulus-Response-Agent

5. Summary and Prospect

Example: 11-Multiplexer

learning a boolean 11-multiplexer [Koza, 1992]



- multiplexer with 8 data- and 3 address lines (state of the address lines indicates active data line)
- $2^{11} = 2048$ possible inputs with 1 corresponding address each
- choose sets of symbols:
  - $\mathcal{T} = \{a_0, a_1, a_2, d_0, \ldots, d_7\}$
  - $\mathcal{F} = \{\text{and}, \text{or}, \text{not}, \text{if}\}$
- fitness function: $f(s) = 2048 - \sum_{i=1}^{2048} e_i$, with e_i being the error for the i-th input

Example: 11-Multiplexer

typical values:

popultaion size $|P| = 4000$

depth of the parse tree on initialization: 6, maximal depth: 17

fitness values in initialization population between 768 and 1280, with a mean of 1063
(with random output: about half of it is actually right, thus the expected value = 1024)

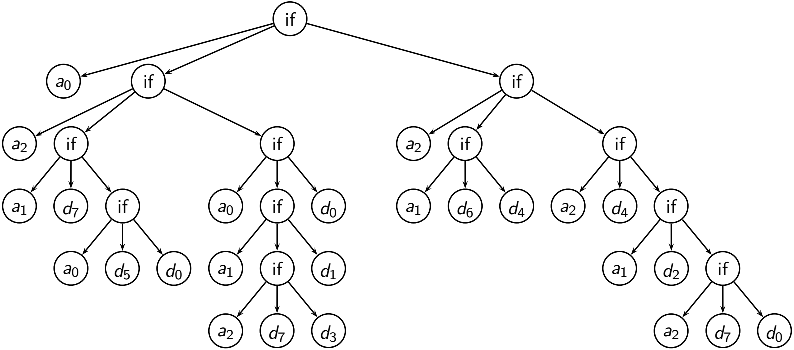
23 expressions with a fitness of 1280, one of them represents a 3-Multiplexer: (if a_0 d_1 d_2)

fitness proportional selection

- 90% (3600) of all individuals are used for crossover
- 10% (400) are taken to the next generation unalteredly

Example: 11-Multiplexer

- after 9 generations: solution with fitness of 2048



rather complicated for humans to understand

can be simplified by editing

Editing

asexual operations on one individual

serves simplification through general and specific rules

general: if function without side effects on constant arguments occurs within the tree, then evaluate this function and replace the subtree with the result

specific: boolean algebra (in this case):

$\neg(\neg A) \to A, \quad (A \wedge A) \to A, \quad (A \vee A) \to A$

de Morgan's Laws, etc.

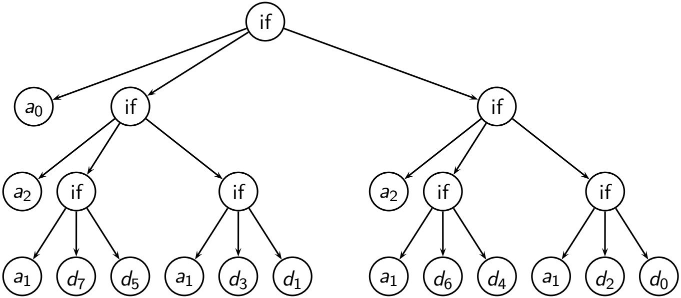
- transformation: e.g. as an operator during GP-search

reduction of bloated individuals usually is a trade-off with diversity of the population

usually: transformation only for result interpretation

11-Multiplexer

best solution truncated by editing:



Example: 11-Multiplexer

best individual in 9th generation reaches best fitness

question: What is the probability for having such an occurence during random search?

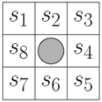
estimated number of all boolean functions:

- How many boolean functions are there for 11 variables?
- Why is this value not sufficient for GPs?
- How many possibilities are there without maximum tree depth?

Stimulus-Response-Agent

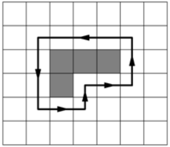
Lerning a Robot Control System [Nilsson, 1998]

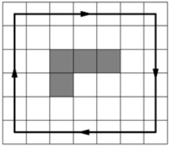
Consider Stimulus-Response-Agenten in Grid-World:



- 8 sensors $s_1, \ldots, s_8$ yield state of the neighboring fields
- 4 actions: go east, go north, go west, go south
- direct deduction of the actions from $s_1, \ldots, s_8$, no memory

task: Circulate an object within a room,
or follow the walls of the room!





Stimulus-Response-Agent

symbol sets:

- $\mathcal{T} = \{s_1, \ldots, s_8, \text{east}, \text{north}, \text{west}, \text{south}, 0, 1\}$
- $\mathcal{F} = \{\text{and}, \text{or}, \text{not}, \text{if}\}$

complete functions, e.g. by

$$(\text{and } x \ y) = \begin{cases} \text{false}, & \text{falls } x = \text{false}, \\ y, & \text{else}. \end{cases}$$

(note: this way a boolean operation can yield an action, too)

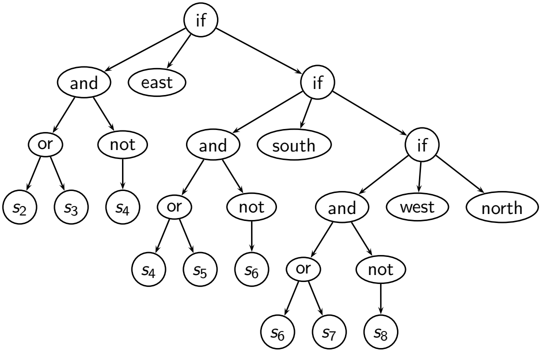
population size $|P| = 5000$, tournament selection with tournament size 5

generating the subsequent population

- 10% (500) candidate solutions are taken to the next generation unchanged
- 90% (4500) candidate solutions are generated by crossover
- <1% candidate solutions are mutated

Stimulus-Response-Agent

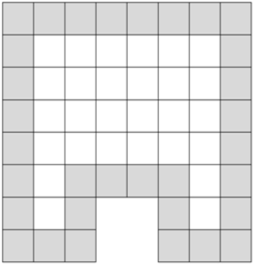
optimal solution created by hand:



It is very unlikely to get exactly this solution.
To keep chromosomes simple, it might be useful to use a penalty term as a measure of the expressions' complexity.

Stimulus-Response-Agent

choose solution candidates by using a test space:



- perfectly operating control unit would make the agent pass the grey labelled fields
- initial field is chosen randomly
- if the action cannot be performed, or if a boolean value is returned, the execution is quit

Agents will be put to 10 different initial fields, and their actions (controlled by the corresponding cromosome) are observed.

Total number of visited gray fields is fitness. (maximum fitness: $10 \cdot 32 = 320$)

Following the Wall

Most of the 5000 programs of generation 0 are useless:

`(and sw ne)`

- just evaluates and terminates
- fitness of 0

`(or east west)`

- sometimes yields `west` thus running one step west
- sometimes ends up besides a wall
- fitness of 5

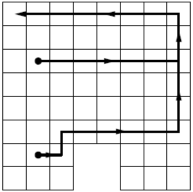
best program achieves fitness of 92

- difficult for testing, because of redundant operators
- path is visualized with 2 differend initial fields on the following slide (east till wall, then north till east or west possible, then trapped in the upper left corner)

best individual of generation no. 0

```
(and (not (not (if  (if  (not s1)
                          (if  s4 north east)
                          (if  west 0 south))
                     (or  (if  s1 s3 s8) (not s7))
                     (not (not north)))))
     (if  (or  (not (and (if  s7 north s3)
                          (and south 1)))
               (or  (or  (not s6) (or s4 s4))
                    (and (if  west s3 s5)
                          (if  1 s4 s4))))
          (or  (not (and (not s3)
                          (if  east s6 s2)))
               (or  (not (if  s1 east s6))
                    (and (if  s8 s7 1)
                          (or  s7 s1))))
          (or  (not (if  (or  s2 s8)
                          (or  0 s5)
                          (or  1 east)))
               (or  (and (or  1 s3)
                          (and s1 east))
                    (if  (not west)
                          (and west east)
                          (if  1 north s8))))))
```

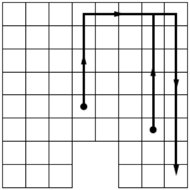
best individual of generation no. 0:



(movements from 2 different initial positions)

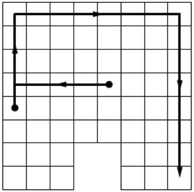
best individual of generation no. 2:

```
(not (and (if  s3
               (if  s5 south east)
               north)
          (and not s4)))
```



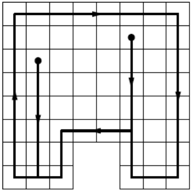
best individual of generation no. 6:

```
(if  (and (not s4)
          (if  s4 s6 s3))
     (or  (if  1  s4 south)
          (if  north east s3))
     (if  (or  (and 0 north)
               (and s4 (if  s4
                            (if  s5 south east)
                            north)))
          (and s4 (not (if  s6 s7 s4)))
          (or  (or  (and s1 east) west) s1)))
```

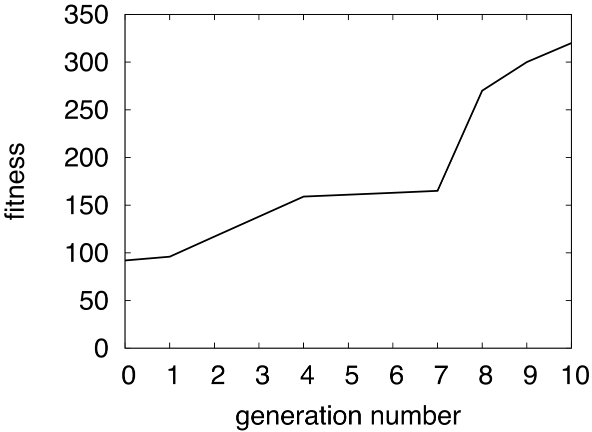


best individual of generation no. 10:

```
(if  (if  (if  s5 0 s3)
          (or  s5 east)
          (if  (or  (and s4 0)
                    s7)
               (or  s7 0)
               (and (not (not (and s6 s5)))
                    s5)))
     (if  s8
          (or north
              (not (not s6)))
          west)
     (not (not (not (and (if  (not south)
                              s5
                              s8)
                         (not s2))))))
```



development of fitness



development of fitness during learning process
(best individual of the current generation)

Outline

Problem of Introns

individuals are growing in size with progressing generation count

- reason: so-called **Introns**:
  - Biology: some strips of DNA carry no information
  - inactive (maybe outdated) strips within one gene that serves no function (*junk DNA*)
- e.g. arithmetical expressions $a + (1 - 1)$ is easy to simplify
- in `if 2 < 1 then ...else ...` the “then“-branch is useless
- changes by operators within active parts of the individual often cause negative effects
- changes within introns often have no effect

$\Rightarrow$ leads to synthetical expansion of the individuals

$\Rightarrow$ portion of active program code decreases - optimization stagnates

Preventing Introns

modify operators:

- **breeding recombination** generates many children from two parents by usnig different parameters, with only the best one going on into the next generation
- **intelligent recombination** chooses crossover points selectively
- **continuous slight changes of the evaluation function** can change constraints thus inactive subprograms (introns) might become active again $\Rightarrow$ works only with non-trivial introns being created by non-changing values

penalty of large individuals
discrimination during selection

Extensions

Encapsulation of automatically defined functions

- potentially promising subexpressions should be protected from being destroyed by crossover or manipulation
- a new function is defined for subexpressions (of a good chromosome), and it's symbol is added to the set $\mathcal{F}$
- number of arguments of the new function = number of (different) arguments of the subtree

Literatur zur Lehrveranstaltung I

Banzhaf, W., Nordin, P., Keller, R. E., and Francone, F. D. (1998).
Genetic Programming — An Introduction: On the Automatic Evolution of Computer Programs and Its Applications.
Morgan Kaufmann Publisher, Inc. and dpunkt-Verlag, San Francisco, CA, USA and Heidelberg, Germany.

Koza, J. R. (1992).
Genetic Programming: On the Programming of Computers by Means of Natural Selection.
MIT Press, Boston, MA, USA.

Nilsson, N. J. (1998).
Artificial Intelligence: A New Synthesis.
Morgan Kaufmann Publishers, Inc., San Francisco, CA, USA.