Synthesis and Induction of Recursive Programs using the Intelligent System $\mathcal{SIPRES}$

Synthesis by means of Induction based on Rewriting and GP

Authors: Edwin Camilo Cubides Garzón, Ph.D.(c)
eccubidesg@unal.edu.co

Jonatan Gómez Perdomo, Ph.D.
jgomezpe@unal.edu.co

Grupo de investigación en vida artificial – Research Group on Artificial Life – (Alife)
Departamento de Ingeniería de Sistemas e Industrial
Facultad de Ingeniería
Universidad Nacional de Colombia

2nd Semester 2019

Outline

Program Synthesis

Definition

Program Synthesis is the task of automatically finding programs from the underlying programming language (automatic programming) that satisfy user intent expressed in some form of constraints described as formal mathematical specification. [1]

Example

- input–output examples (induction)
- demonstrations
- partial programs
- assertions
- natural language

Inductive Logic Programming (ILP) $E^+ \uplus E^- \uplus BKG \Rightarrow H$

Prof. Stephen Muggleton (1991)

Inductive Logic Programming (ILP) is a research area formed as the intersection of Machine Learning and Logic Programming [2]. ILP uses logic programming as a uniform representation for examples, background knowledge and theories

Given an encoding of the known background knowledge and a set of examples represented as a logical database of facts, an ILP system will provide a theory as a logic program (hypothesis) which entails all the positive and none of the negative examples.

$$\text{Schema:} \left\{ \begin{array}{c} \text{Positive examples } (E^+) \\ \uplus \\ \text{Negative examples } (E^-) \\ \uplus \\ \text{Background knowledge } (BKG) \end{array} \right. \Longrightarrow \text{Hypothesis } (H)$$

General Algorithm to Evolve Individuals

Algorithm 1 $\textsc{Evolutionary\_Algorithm}(f, \mu, \text{terminationCondition})$

$t = 0$
$P_0 = \textsc{InitPopulation}(\mu)$
$\text{evaluate}(P_0, f)$
while $(\text{terminationCondition}(t, P_t, f) = \mathsf{false})$ **do**
  $newIndividuals = \textsc{Generate}\left(P_t, f, \textsc{Select\_Function}\right)$
  $P_{t+1} = offspring = \textsc{Replacement}(P_t, newIndividuals, f)$
  $\text{evaluate}(P_{t+1}, f)$
  $t = t + 1$
end while
return P_t

The parameter $\textsc{Select\_Function}$ is a function used for selecting the parents to generate new individuals using the genetic operators.

HaEa: Hybrid Adaptive Evolutionary Algorithm I

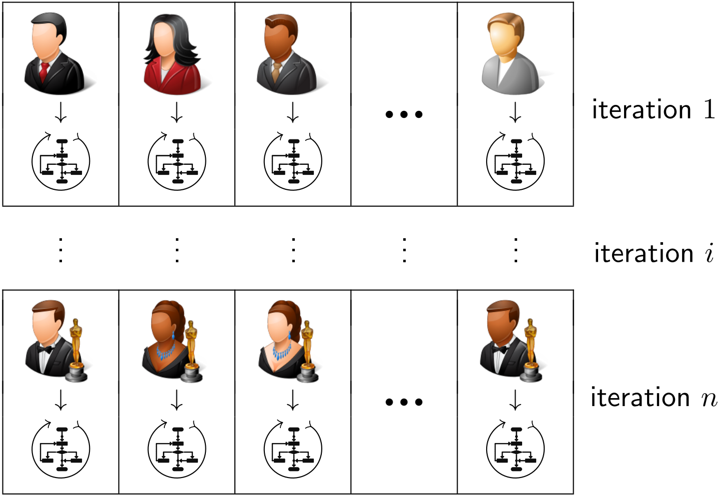
Algorithm 2 Hybrid Adaptive Evolutionary Algorithm

HaEa(fitness, μ, terminationCondition)

$t = 0$
$P_0 = \text{INITPOPULATION}(\mu)$
evaluate(P_0, fitness)
while (terminationCondition(t, P_t, fitness) is false) **do**
  $P_{t+1} = \varnothing$
  for each ind $\in P_t$ **do**
    ...
    ...
  end for
  $t = t + 1$
end while

HaEa: Hybrid Adaptive Evolutionary Algorithm II

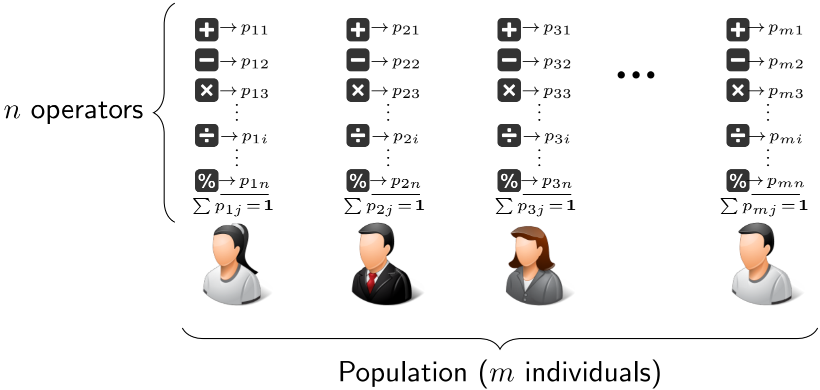
Main cycle in HaEa



for each ind $\in P_t$ **do**
...
end for

HaEa: Hybrid Adaptive Evolutionary Algorithm III

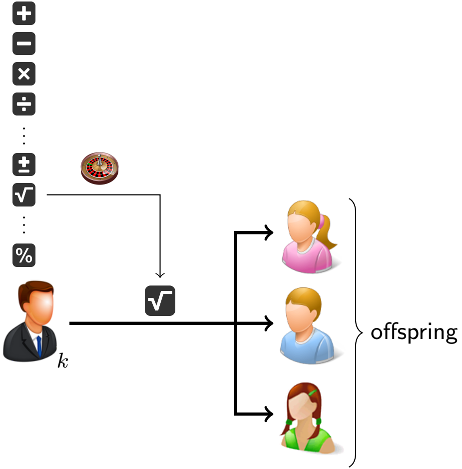
Population and rates operators



rates = extrac_rates_oper(ind)

HaEa: Hybrid Adaptive Evolutionary Algorithm IV

Unary operators and offspring

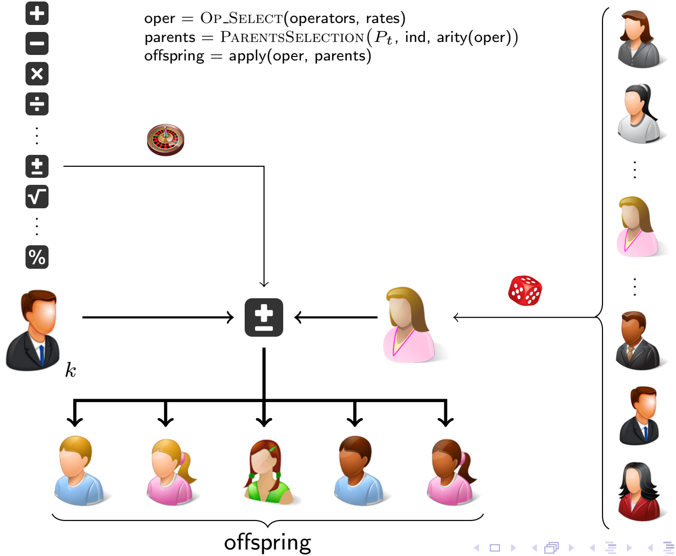


oper $=$ OP_SELECT(operators, rates)
parents $=$ PARENTSSELECTION$(P_t$, ind, arity(oper)$)$
offspring $=$ apply(oper, parents)

HaEa: Hybrid Adaptive Evolutionary Algorithm V

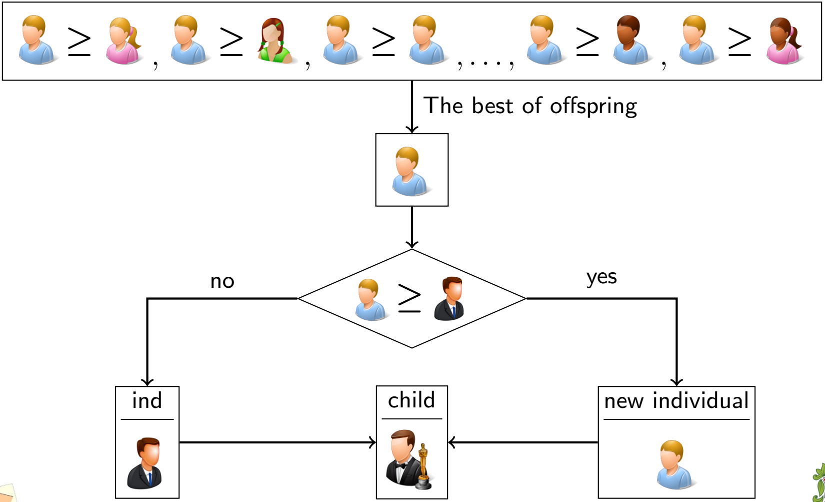
binary operators and offspring

oper = OP_SELECT(operators, rates)
parents = $\text{PARENTSSELECTION}\left(P_t,\right.$ ind, arity(oper)$\left.\right)$
offspring = apply(oper, parents)



HaEa: Hybrid Adaptive Evolutionary Algorithm VI

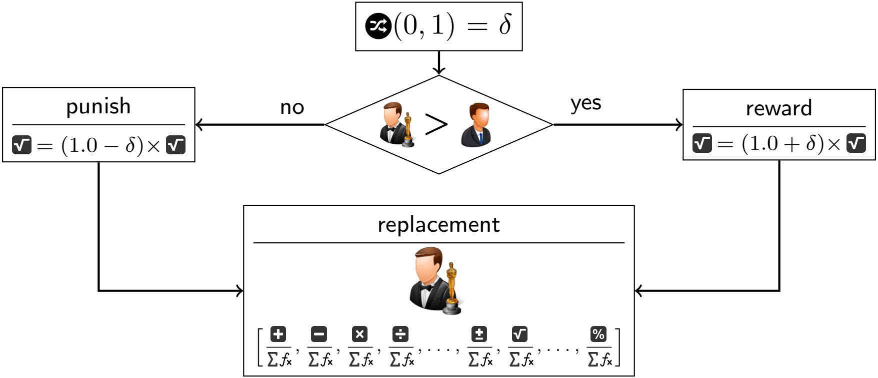
Steady state replacement



child $=$ Best(offspring, ind)

HaEa: Hybrid Adaptive Evolutionary Algorithm VII

Reward, punish and normalization of rates



δ = random(0, 1) // *learning rate*
if (fitness(child) > fitness(ind)) **then**
    rates[oper] = (1.0 + δ) ∗ rates[oper] // *reward*
else
    rates[oper] = (1.0 − δ) ∗ rates[oper] // *punish*
end if
normalize_rates(rates)
set_rates(child, rates)
$P_{t+1} = P_{t+1} \cup$ {child}

Representation in $\mathcal{SIPRES}$-algorithm I

Phenotype

Phenotype: Phenotype of Individuals in $\mathcal{SIPRES}$ are represented as a list of directed equations separated by semicolons, thus:

$$\mathcal{P} \equiv \{e_1; e_2; e_3; \cdots ; e_{n-1}; e_n\}$$

Example

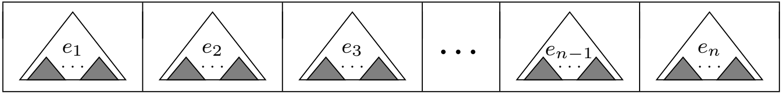
```
{yinyang(0) = true; yinyang(s(s(N))) = yinyang(N); yinyang(1) = false}
```

```
{maltese_cross(N,0) = N; maltese_cross(N,s(M)) = maltese_cross(s(N),M)}
```

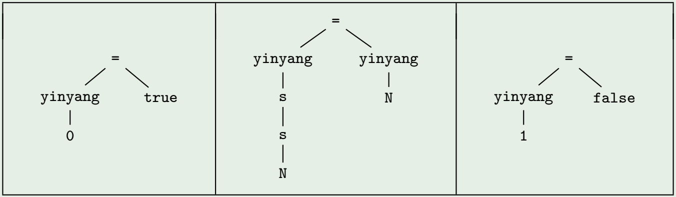
Representation in $\mathcal{SIPRES}$-algorithm II

Genotype

Genotype: Genotype of Individuals or Chromosomes in $\mathcal{SIPRES}$ are represented as a list or forest of the arborescent representation (genes) of the equations $e_1, e_2, e_3, \ldots, e_{n-1}, e_n$, thus:

$\mathcal{P} \equiv$ 

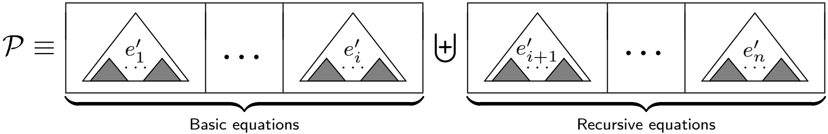
Example



Representation in $\mathcal{SIPRES}$-algorithm III

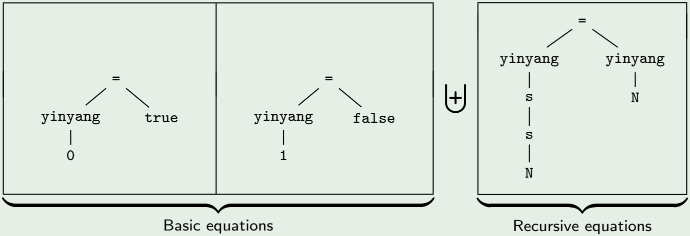
Genotype

Internally a program is also represented as the disjoint union of two sets, namely, the basic equations set and the recursive equations set.



where $\{e_1, e_2, e_3, \ldots, e_{n-1}, e_n\} = \{e'_1, e'_2, \ldots, e'_i, e'_{i+1}, \ldots, e'_{n-1}, e'_n\}$

Example



Initial population in $\mathcal{SIPRES}$-algorithm I

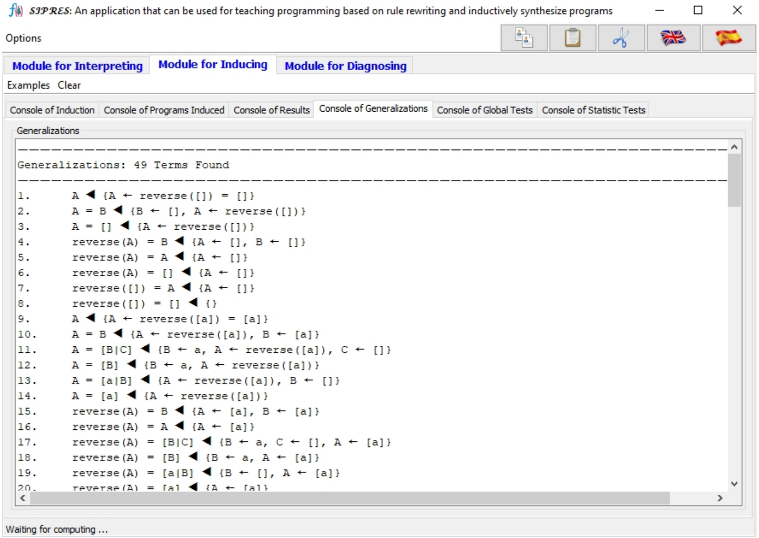
Positive Basic Examples (E^{+}) – Training Dataset to Generalize

```
reverse([]) = []
reverse([a]) = [a]
```

Positive Extra Examples (E^{++}) – Training Dataset to Filter

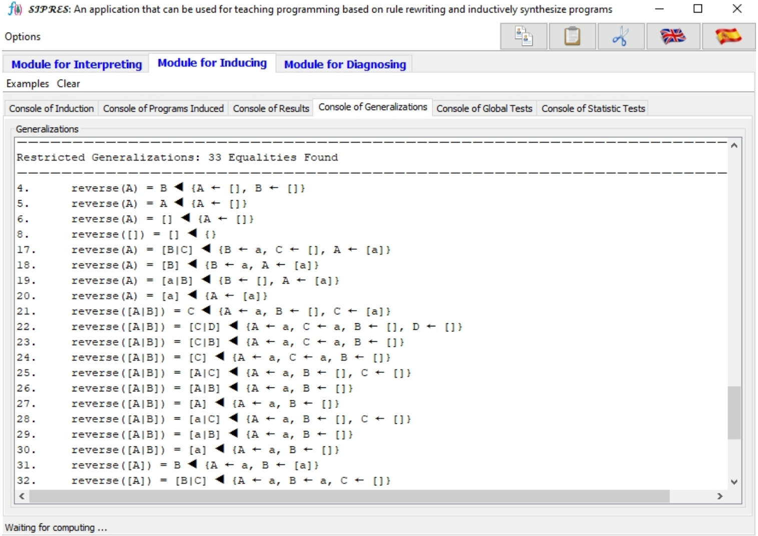
```
reverse([x,y]) = [y,x]
reverse([x,y,z]) = [z,y,x]
reverse([w,x,y,z]) = [z,y,x,w]
reverse([0,_,0]) = [0,_,0]
```

Initial population in *SIPRES*-algorithm II



49 terms

Initial population in $\mathcal{SIPRES}$-algorithm III



49 term $\rightsquigarrow$ 33 equalities

Initial population in $\mathcal{SIPRES}$-algorithm IV

n programs

	Basic Equations	Recursive Equations
$\{e_{0 \bmod m}; random_0(E)\}$	$e_{0 \bmod m}$	$random_0(E)$
$\{e_{1 \bmod m}; random_1(E)\}$	$e_{1 \bmod m}$	$random_1(E)$
$\vdots$	$\vdots$	$\vdots$
$\{e_{i \bmod m}; random_i(E)\}$	$e_{i \bmod m}$	$random_i(E)$
$\vdots$	$\vdots$	$\vdots$
$\{e_{(n-1) \bmod m}; random_{(n-1)}(E)\}$	$e_{(n-1) \bmod m}$	$random_{(n-1)}(E)$

n individuals

$E =$ set of generalizations, $e_i \in E$, $m = |E|$,

$n = \max\{m, \text{Min. size population}\}$

Initial population in SIPRES-algorithm V

Reparation of Equations

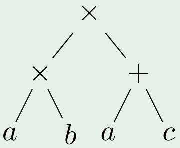
```
1         reverse(A) = reverse(A); reverse([A|B]) = [reverse(B)|A]
2         reverse(A) = A; reverse(A) = reverse(A)
3         reverse(A) = [A|A]; reverse([A|B]) = reverse(B)
4         reverse(A) = []; reverse(A) = [A|reverse(A)] <---
5         reverse([A|A]) = []; reverse([a|A]) = [a|reverse(A)] <---
6         reverse([]) = []; reverse([a|A]) = [a]
7         reverse(A) = [reverse(A)|A]; reverse([A]) = [A|reverse(A)]
8         reverse(A) = [reverse(A)]; reverse([A|B]) = [B|reverse(A)]
9         reverse(A) = [a|reverse(A)]; reverse(A) = [A|A]
10         reverse(A) = [a]; reverse(A) = []
11         reverse([A|B]) = reverse(A); reverse([A|B]) = [a]
12         reverse([A|B]) = [A|reverse(B)]; reverse([A|B]) = [reverse(A)]
13         reverse([A|B]) = [B|reverse(A)]; reverse([a|A]) = [a]
   ...
   ...
```

Initial population in Classic GP

Full: Length of all branches equal to $l > 0$.

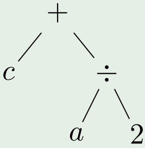
Grow: Length of all branches of depth less than or equal to $l > 0$.

Example (Full: Depth $l = 2$)



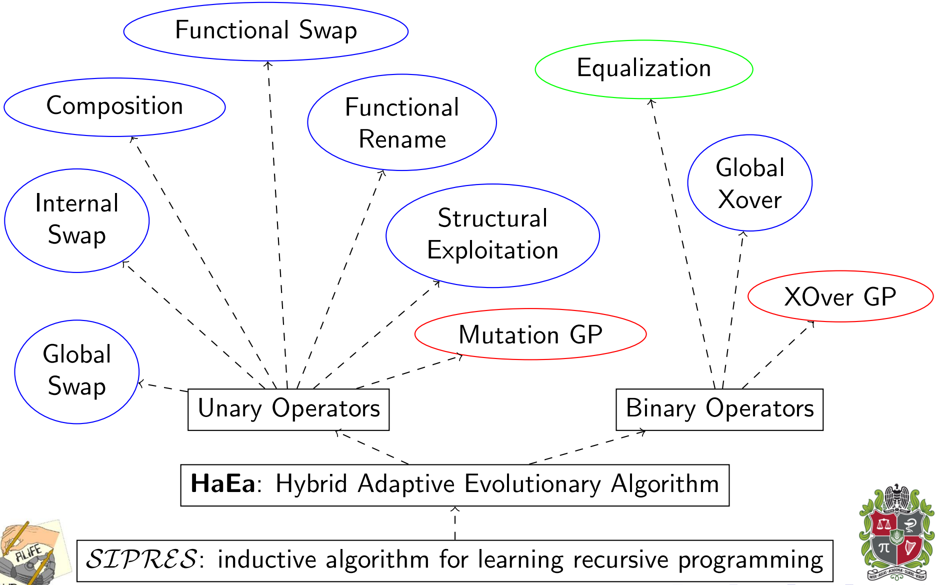
$(a \times b) \times (a + c)$

Example (Grow: Depth $0 < l \leq 2$)



$c + (a \div 2)$

Operators in *SIPRES*-algorithm



Unary operator: Global Swap

Example

$$p = \{\texttt{sum\_n(N) = N};\ \texttt{sum\_n(s(N)) = sum(s(N),sum\_n(N))}\}$$

$$\Downarrow$$

$$p' = \{\texttt{sum\_n(s(N)) = sum(s(N),sum\_n(N))};\ \texttt{sum\_n(N) = N}\}$$

Unary operator: Internal Swap

Example

$$p = \{\texttt{prod(N,0) = 0;}\ \texttt{prod(s(M),N) = sum(N,prod(N,M))}\}$$

$$\Downarrow$$

$$p' = \{\texttt{prod(N,0) = 0;}\ \texttt{prod(N,s(M)) = sum(N,prod(N,M))}\}$$

Unary operator: Functional Swap

Example

$$p = \{\texttt{prod(N,0) = 0};\ \texttt{prod(s(M),N) = prod(N,sum(N,M))}\}$$

$$\Downarrow$$

$$p' = \{\texttt{prod(N,0) = 0};\ \texttt{prod(s(M),N) = sum(N,prod(N,M))}\}$$

Unary operator: Functional Rename

Example

$$p = \{\texttt{sum(N,0) = N};\ \texttt{sum(s(N),M) = s(sum(N,M))}\}$$

$$\Downarrow$$

$$p' = \{\texttt{sum(N,0) = N};\ \texttt{sum(N,s(M)) = s(sum(N,M))}\}$$

Binary operator: Global XOver

Example

$$p_1 = \{\texttt{sum\_n(N) = N};\ \texttt{sum\_n(s(N)) = sum(N,sum\_n(N))}\}$$

$$p_2 = \{\texttt{sum\_n(s(N)) = s(sum(N,sum\_n(N)))};\ \texttt{sum\_n(0) = 1}\}$$

$$\Downarrow$$

$$p_1' = \{\texttt{sum\_n(s(N)) = s(sum(N,sum\_n(N)))};\ \texttt{sum\_n(N) = N}\}$$

$$p_2' = \{\texttt{sum\_n(s(N)) = sum(N,sum\_n(N))};\ \texttt{sum\_n(0) = 1}\}$$

Binary operator: Equalization I

Example

$$p_1 = \{\texttt{sum\_n(s(A)) = sum(s(A),A)}\}$$
$$p_2 = \{\texttt{sum\_n(A) = A}\}$$

$$\Downarrow$$

$$e_1 = \texttt{sum\_n(s(A)) = sum(s(A),A)} \qquad (receptor)$$
$$e_2 = \texttt{sum\_n(A) = A} \qquad (emitter)$$

$$\Downarrow$$

$$e_1 = \texttt{sum\_n(s(A)) = sum(s(A),A)}$$
$$\widehat{e_2} = \texttt{sum\_n(N) = N}$$

Binary operator: Equalization II

Example

$$e_1 = \texttt{sum\_n(s(A)) = sum(s(A),A)}$$
$$\widehat{e_2} = \texttt{sum\_n(N) = N}$$

$$\Downarrow$$

```
sum_n(s(A)) = sum_n(N)
sum_n(s(A)) = sum(sum_n(N),A)
sum_n(s(A)) = sum(s(sum_n(N)),A)
sum_n(s(A)) = sum(s(A),sum_n(N))
```

$$\Downarrow$$

Binary operator: Equalization III

Example

```
sum_n(s(A)) = sum_n(N)
sum_n(s(A)) = sum(sum_n(N),A)
sum_n(s(A)) = sum(s(sum_n(N)),A)
sum_n(s(A)) = sum(s(A),sum_n(N))
```

$$\Downarrow$$

```
sum_n(s(A)) = sum_n(A)
sum_n(s(A)) = sum(sum_n(A),A)
sum_n(s(A)) = sum(s(sum_n(A)),A)
sum_n(s(A)) = sum(s(A),sum_n(A))
```

$$\Downarrow$$

Binary operator: Equalization IV

Example

$$p'_1 = \{\texttt{sum\_n(s(A)) = sum(s(A),A)};\ \texttt{sum\_n(A) = A};\ \texttt{sum\_n(s(A)) = sum\_n(A)}\}$$

$$p'_2 = \{\texttt{sum\_n(s(A)) = sum(s(A),A)};\ \texttt{sum\_n(A) = A};\ \texttt{sum\_n(s(A)) = sum(sum\_n(A),A)}\}$$

$$p'_3 = \{\texttt{sum\_n(s(A)) = sum(s(A),A)};\ \texttt{sum\_n(A) = A};\ \texttt{sum\_n(s(A)) = sum(s(sum\_n(A)),A)}\}$$

$$p'_4 = \{\texttt{sum\_n(s(A)) = sum(s(A),A)};\ \texttt{sum\_n(A) = A};\ \texttt{sum\_n(s(A)) = sum(s(A),sum\_n(A))}\}$$

Unary operator: Composition

Emitter: Basic equations of BKG and repaired primitive generalizations of them.

Receptor: Equations of E^+.

Example

```
palindrome([]) = true
palindrome([0]) = true
palindrome([x,x]) = true
palindrome([x,y]) = false
palindrome([x,y,x]) = true
palindrome([x,x,x]) = true
palindrome([x,y,z]) = false
palindrome([x,y,y,x]) = true
```

```
equals(A,A) = true
equals(A,B) = false
append([],L) = L
append([H|T],L) = [H|append(T,L)]
reverse([]) = []
reverse([H|T]) = append(reverse(T),[H])
```

```
equals(A,B) = A
append(A,B) = A
reverse(A) = A
```

```
palindrome(A) = equals(reverse(A),A)
```

Unary operator: Structural Exploitation

Example

Background knowledge:
```
max(0,A) = A;
max(A,0) = A;
max(s(A),s(B)) = s(max(A,B));
min(s(N),s(M)) = s(min(N,M));
min(N,M) = 0
```

Primitive functors: `max(B,A)`, `min(A,B)`, `[]`, `[A]`, `[A|B]`.

$$p = \{\texttt{insert(A,[B|C]) = [A|insert(B,C)]}\}$$

$$\Downarrow \texttt{max(B,A)}$$

$$p' = \{\texttt{insert(A,[B|C]) = [A|insert(max(B,A),C)]}\}$$

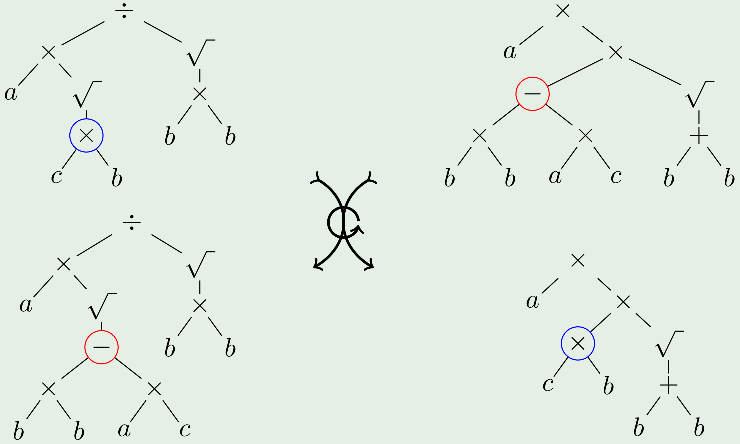
$$\Downarrow \texttt{min(A,B)}$$

$$p'' = \{\texttt{insert(A,[B|C]) = [min(A,B)|insert(max(B,A),C)]}\}$$

Binary operator: XOver GP

Crossover: given a couple of programs, a subtree of each program is randomly selected and these subtrees are swapped.

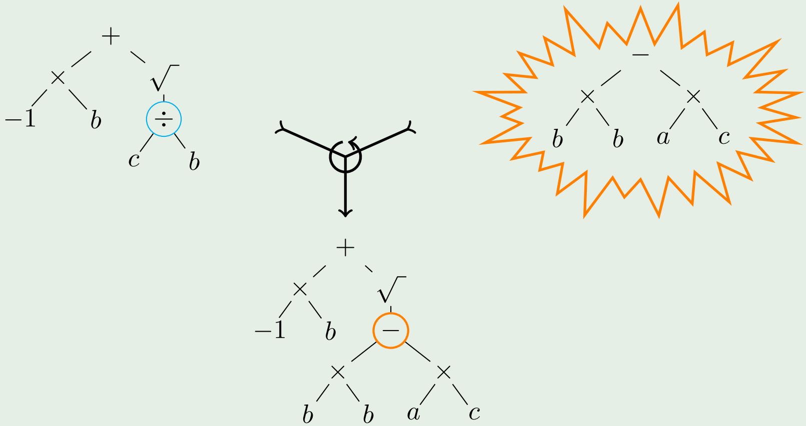
Example



Unary operator: Mutation GP

Mutation: given a program, a subtree of this program is randomly selected and it is replaced by a new randomly generated tree.

Example



Fitness function

Definition (Covering)

The *covering* (Cov) of a program p is defined as the set of basic and extra positive examples that p can deduce, in others words

$$Cov(p) = \{e \in E^{+} \cup E^{++} : p \vdash e\},$$

and the *fitness function* or *covering factor* ($CovF^{+}(p)$) is defined as the cardinal of the set $Cov(p)$ divided by the cardinal of the set of positive examples, thus

$$CovF^{+}(p) = \frac{|Cov(p)|}{|E^{+} \cup E^{++}|}$$

whence $CovF^{+}(p) \in [0, 1]$.

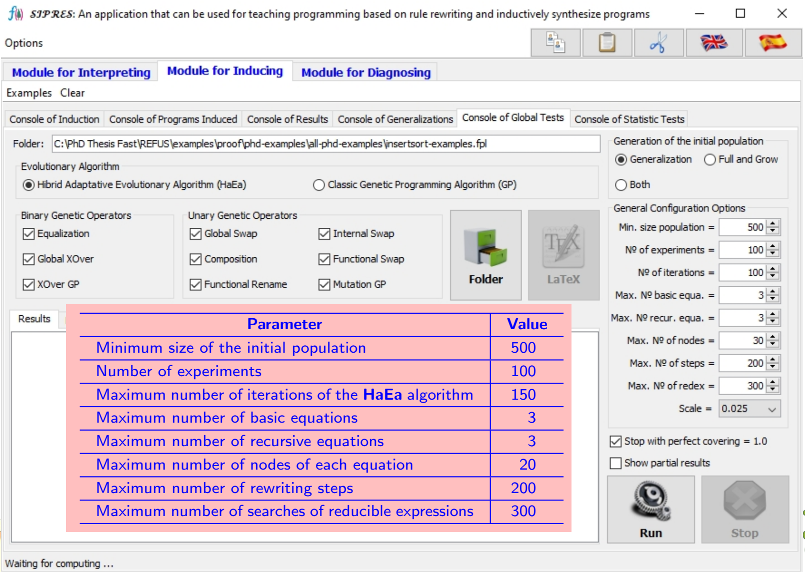
Outline

1. Synthesis and Induction of Recursive Programs using $\mathcal{SIPRES}$

2. Experiments & Results

3. The $\mathcal{FUNICO}$ Index

Experiments **HaEa** I: Parameters of Configuration



Parameter	Value
Minimum size of the initial population	500
Number of experiments	100
Maximum number of iterations of the **HaEa** algorithm	150
Maximum number of basic equations	3
Maximum number of recursive equations	3
Maximum number of nodes of each equation	20
Maximum number of rewriting steps	200
Maximum number of searches of reducible expressions	300

Comparison of HaEa, FLIP and GP I

FLIP: A Functional Logic Inductive Programming System

`http://users.dsic.upv.es/~flip/flip/index.html`

Program	**HaEa**	FLIP	GP
ack	✓		
combination	✓		
diff	✓		✓
double	✓	✓	✓
even	✓	✓	✓
factorial	✓	✓	✓
fibonacci	✓		
geq	✓	✓	✓
greater	✓	✓	✓
hanoi	✓		✓
leq	✓	✓	✓
less	✓	✓	✓

Program	**HaEa**	FLIP	GP
max	✓	✓	✓
min	✓	✓	✓
mod2	✓	✓	✓
mod3	✓	✓	✓
odd	✓	✓	✓
pow	✓		✓
pow-ext	✓		✓
prod	✓	✓	✓
sum	✓	✓	✓
sum-prod	✓	✓	✓
tetration	✓		✓
tribonacci	✓		
triple	✓	✓	✓

Program	**HaEa**	FLIP	GP
and	✓		✓
iff	✓		✓
not	✓	✓	✓
or	✓	✓	✓
then	✓		✓
xor	✓		✓
enjoy sport	✓	✓	✓
if	✓	✓	✓
play-tennis	✓	✓	

Comparison of HaEa, FLIP and GP II

FLIP: A Functional Logic Inductive Programming System

Program	**HaEa**	FLIP	GP
allequals	✓		✓
append	✓	✓	✓
consec	✓	✓	✓
count	✓		
first	✓	✓	✓
get	✓		✓
insert	✓		✓
insertsort	✓		
last	✓	✓	✓
length	✓	✓	✓
maxlist	✓	✓	✓

Program	**HaEa**	FLIP	GP
member	✓	✓	✓
minlist	✓	✓	✓
palindrome	✓		✓
pop	✓	✓	✓
prefix	✓		
prodlist	✓	✓	✓
push	✓	✓	✓
remainder	✓		✓
remove	✓	✓	
reverse	✓	✓	
selectsort	✓		
suffix	✓		✓
sumlist	✓		

Outline

1. Synthesis and Induction of Recursive Programs using $\mathcal{SIPRES}$
2. Experiments & Results
3. The $\mathcal{FUNICO}$ Index

Functional Induction Complexity Index I

$\mathcal{FUNICO}$

Definition

Let $\mathcal{R}$ be a TRS, with $\mathcal{R} = \{e_1; e_2; \cdots; e_n\} \cup bkg$, where bkg is the background knowledge for the TRS $\mathcal{R}$. $\mathcal{FUNICO}(\mathcal{R})$ could be computed using following functions:

$$\mathcal{FUNICO} : \mathcal{TRS} \to \mathbb{N}$$

$$\{e_1; e_2; \cdots; e_n\} \cup bkg \mapsto \sum_{i=1}^{n} \mathcal{FUNICO}(e_i) + |\text{basic}(bkg)|$$

$$\mathcal{FUNICO} : \mathcal{E} \to \mathbb{N}$$

$$(s = t) \mapsto \begin{cases} 1, & \text{if } \nexists\, u \in O(t) \text{ such that } t[u] \in \mathcal{F}; \\ \mathcal{FUNICO}(s) + \mathcal{FUNICO}(t), & \text{otherwise.} \end{cases}$$

Functional Induction Complexity Index II

$\mathcal{FUNICO}$

Definition (continuation)

$$\mathcal{FUNICO} : \mathcal{T}(\Sigma, \mathcal{V}) \to \mathbb{N}$$

$$(t) \mapsto \begin{cases} 1, & \text{if } (t \in \mathcal{V}) \vee \big(arity(t) = 0\big); \\ 1 + \mathcal{FUNICO}(t_1) + & \text{if } \Big[\big(t = f(t_1, t_2)\big) \wedge \\ \mathcal{FUNICO}(t_2), & \quad \big(f(t_1, t_2) = f(t_2, t_1)\big)\Big] \vee \\ & \quad \big(t = \bullet(t_1, t_2)\big); \\ \mathcal{FUNICO}(t_1), & \text{if } \big(t = \mathsf{s}(t_1)\big) \wedge \\ & \quad \big(\exists\, u \in O(t_1) : t[u] \in \mathcal{F}\big); \\ 0, & \text{if } \big(t = \mathsf{s}(t_1)\big) \wedge \\ & \quad \big(\nexists\, u \in O(t_1) : t[u] \in \mathcal{F}\big); \\ arity(f) + \sum_{i=1}^{n} \mathcal{FUNICO}(t_i), & \text{if } t = f(t_1, t_2, \ldots, t_n). \end{cases}$$

Square Logistic Function and Mean Squared Error (MSE)

Definition (Square Logistic Function)

The Logistic Function or Logistic Curve is a Sigmoid Curve such that is defined as:

$$f(x) = \frac{L}{1 + Ce^{Ax^2}}$$

where L is the threshold (upper bound of the function) and the parameters C and A are the values to estimate (generally with linearization).

Definition (Mean Squared Error (MSE))

The Mean Squared Error can be calculate as:

$$MSE(f) = \left(\frac{1}{N} \sum_{k=1}^{N} \left| f(x_k) - y_k \right|^2 \right)^{1/2}$$

HaEa + Generalization + 500

Curve fitting using Logistic Function

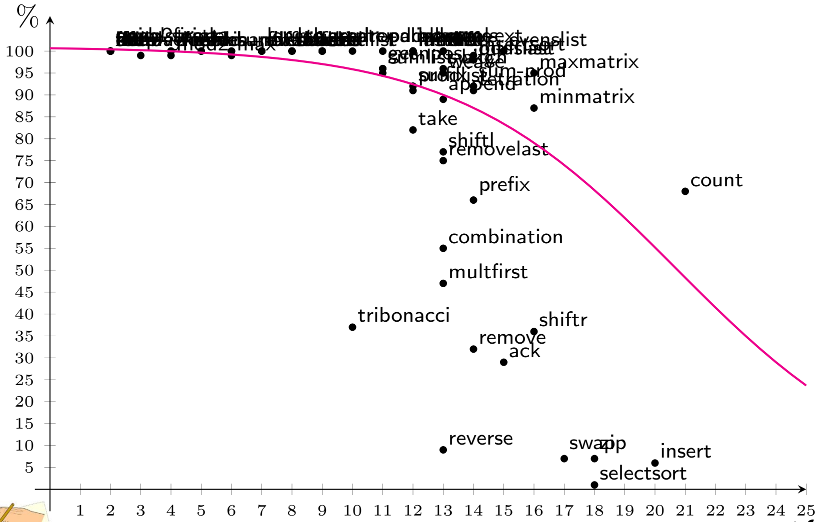


Figure: Time: 11h-29m-6s. Mean Squared Error (MSE) = 21.852945201393535

HaEa + Generalization + 500

Curve fitting using Square Logistic Function

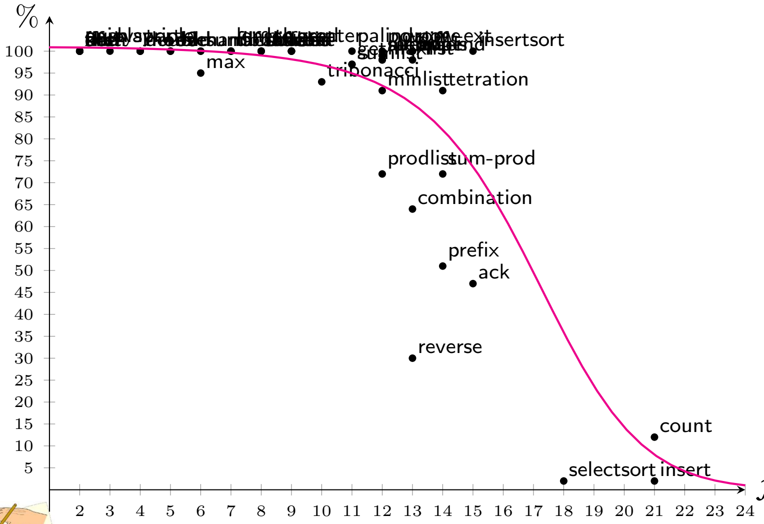


Figure: Time: 19h-42m-8s. Mean Squared Error (MSE) = 12.5460190135894

References I

- Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh, *Program synthesis*, Foundations and Trends in Programming Languages **4** (2017), no. 1-2, 1–119.
- Stephen H. Muggleton, *Inductive Logic Programming*, New Generation Computing **8** (1991), 295–318.

Thank you!

Questions?