GECCO 2005

Grammatical Evolution Tutorial

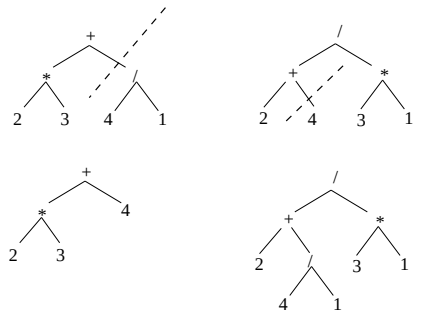
Biocomputing and Developmental Systems
Department of Computer Science & Information Systems
University of Limerick, Ireland.

Conor.Ryan@ul.ie
Hammad.Majeed@ul.ie
Michael.Oneill@ul.ie

Contents

Issues with GP

- Function/terminal set must have "closure"



- Single types only
  - No node context information
- Trees grow, or "bloat"

Biological Phenomena

- No simple one to one mapping
  - Genes produce proteins
  - Proteins combine to create phenotype
- Linear strings
  - Genomes are always held on strings
- Unconstrained search
  - Repair not performed

Grammatical Evolution

- Grammatical Evolution (GE)
  - GA to evolve programs
  - Morphogenetic Effect:
    - Genotype mapped to phenotype
  - Phenotype is a compilable program
- Genome governs mapping of a BNF grammar definition to the program

- Here genome (a binary string) is mapped to compilable C code
- Can potentially evolve programs in any language, with arbitrary complexity
- Any structure than be specified with a grammar, e.g. graphs, neural networks, etc.

Language Definition

- Backus Naur Form (BNF)
  - Notation for expressing a languages grammar as Production Rules
- BNF Grammar consists of the tuple $<$ T,N,P,S $>$ where
  - T is Terminals set
  - N is Non-Terminals set
  - P is Production Rules set
  - S is Start Symbol (a member of N)
- BNF Example

$$T = \{Sin, Cos, Tan, Log, +, -, /, *, X, (,)\}$$

$$S =< expr >$$

BNF Definition

$$N = \{expr, op, pre\_op\}$$

- And P can be represented as:

```
(1) <expr> ::= <expr> <op> <expr>     (A)
             | ( <expr> <op> <expr> ) (B)
             | <pre-op> ( <expr> )    (C)
             | <var>                  (D)

(2) <op> ::= +                        (A)
           | -                        (B)
           | /                        (C)
           | *                        (D)

(3) <pre-op> ::= Sin                  (A)
               | Cos                  (B)
               | Tan                  (C)
               | Log                  (D)

(4) <var> ::= X                       (A)
```

- A Genetic Algorithm is used to control choice of production rule

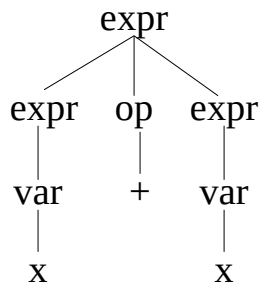
Related GP Systems

Name	Genome	Representation
Koza	Tree	Direct
Banzhaf et al	Linear	Direct
Gruau	Tree	Graph Grammar
Whigham	Tree	Derivation Tree
Wong & Leung	Tree	Logic Grammars
Paterson	Linear	Grammar

Repair mechanisms..

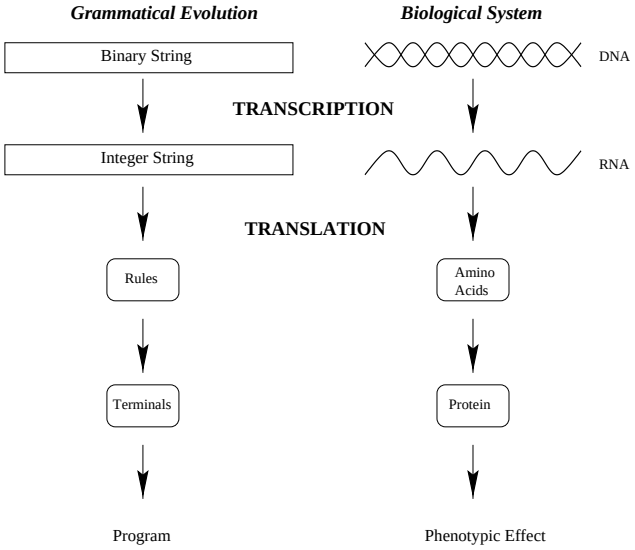
- **Koza - none needed**
- **Banzhaf - required for syntactically legal individuals**
- **Gruau - none needed**
- **Whigham - all crossovers subject to repair**
- **Wong & Leung - all crossovers subject to repair**
- **Paterson - under/over-specification.**

Repair



Grammatical Evolution

- **In contrast GE uses**
  - **BNF - Paterson/Whigham/Wong etc.**
  - **Variable Length Linear Chromosomes - Koza/Gruau/Banzhaf**
  - **Genome encodes pseudo-random numbers**
  - **Degenerate Genetic Code**
    - **Several genes map to same phenotype**
  - **Wrap individuals**
- **Use 8 bit codons**
  - **Each codon represents at least one Production Rule**
  - **Gene contains many codons**
- **Pseudo-random numbers determine what production rule will be used**
- **Expression of a** $Codon$ **results in an** $Amino\ Acid$
  (choice in the derivation sequence)
  - $Amino\ acids$ **can combine to form a functional protein (i.e. Terminals such as** $+$**,** X **or** Sin**, can combine)**



Example Individual

- **To complete BNF definition for a function written in a subset of C we include.....**

```
<func> ::= <header>

<header> ::= float symb(float X) { <body> }

<body> ::= <declarations><code><return>

<declarations> ::= float a;

<code> ::= a = <expr>;

<return> ::= return (a);
```

- **Note implementation details.....**
  - **Function is limited to a single line of code**
    - **If required can get GE to generate multi-line**

functions.....modify `<code>` **to read e.g.:**

```
<code> ::= <line>;
         | <line>; <code>
```

- **In this subset of C all individuals of the form**

```
float symb(float x)
{
  float a;
  a = <expr>;
  return(a);
}
```

- **Only** `<expr>` **will be evolved**
- **Each non-terminal is mapped to a terminal before any others undergo a mapping process**
- **Given the individual**

220	203	51	123	2	45

.....what will happen?

- `<expr>` **has 4 production rules to choose from**

```
(1) <expr> ::= <expr> <op> <expr>     (A)
             | ( <expr> <op> <expr> ) (B)
             | <pre-op> ( <expr> )    (C)
             | <var>                  (D)
```

  - **Taking first codon 220 we get** $220\ MOD\ 4 = 0$
  - **Gives** `<expr><op><expr>`
- **Next choice for the first** `<expr>`
  - **Taking next codon 203 we get** $203\ MOD\ 4 = 3$
  - **Gives** `<var><op><expr>`

- <var> involves no choice
  - Mapped to X...only one production
  - Now have X<op><expr>

220	203	51	123	2	45

- Read next codon to choose <op>
  - Next is third codon , value 51, so get $51\ MOD\ 4 = 3$
  - Now have X*<expr>
- Next choice for <expr>
  - Next codon is 123 so get $123\ MOD\ 4 = 3$
  - Now have X*<var>

- Again <var> involves no choice
  - Finally we get X*X
- The extra codons at end of genome are simply ignored in mapping the genotype to phenotype

Mapping Process

- No simple one to one mapping in GE
- Mapping Process to generate programs
  - Separate Search and Solution Spaces
  - Ensure validity of individuals
  - Remove language dependency
  - Maintain diversity

Genetic Operators

- Perform unconstrained Evolutionary Search
- GE employs standard operators of Genetic Algorithms
- No reason why GE can't be used with other search algorithms
  - Any algorithms that can operate on binary strings will work
    - Hill climbing, Simulated Annealing etc.

Genetic Code Degeneracy

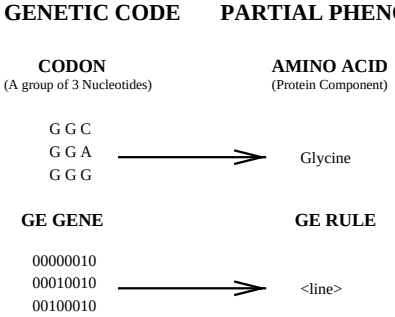


Figure 1: The Degenerate Genetic Code

Genetic Code Degeneracy

- **Neutral Mutations**
  - **Mutations having no effect on Phenotype Fitness**
- **Help preserve individual validity**
- **Gradual accumulation of mutations without harming functionality**
  - **Revisit later**

Initialisation

- **Individuals are strings of random number**
  - **No guarantee that they will terminate**
  - **Individuals can be** *very* **short.**

```
<expr> ::= <expr> <op> <expr>
         | ( <expr> <op> <expr> )
         | <pre-op> ( <expr> )
         | <var>
```

- **Production** `<expr>-><var>` **always leads to termination**
- `<expr>` **is the start symbol**
  - **On average, a quarter of all individuals are just one point**

Sensible Initialisation

- **Generate a spread of individual sizes.**
  - **Based on** *Ramped Half and Half* **initialisation in GP**
    - **For all tree depths from 2 to maximum size**
    - **Generate an equal number of trees of that size**
    - **Use** *full* **for 50%**
    - **Use** *grow* **for 50%**
- **Similar in GE, but generate** *derivation trees* **of equivalent size**

Sensible Initialisation - 2

- Record which number choice was made for each step
- Perform an "unmod" on list of choices
  - Produce a number between 0 and 255 that produces the original number when moded by the number of choices for that production rule
- Ensures that *all* individuals are valid
- Reduces the number of clones (easier to detect)
- Eliminates single point individuals (if desired)

Issues with GE?

- Effect of mutation?
- Degenerate Genetic Code
- Crossover
  - Ripple
  - Homologous
  - Headless

Ripple Crossover - 1

- Analyse 1-point crossover in terms of derivation & syntax trees

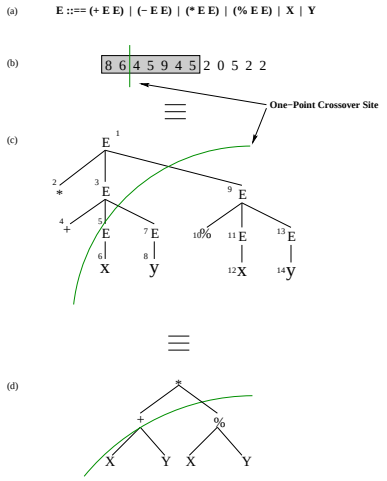


Figure 2: The ripple effect of one-point crossover illustrated using an example GE individual represented as a string of codon integer values (b) and its equivalent derivation (c) and parse trees (d). The codon integer values in (b) represent the rule number to be selected from the grammar outlined in (a), with the part shaded gray corresponding to the values used to produce the trees in (c) and (d), the remaining integers are introns. Figure 3 shows the resulting spine with ripple sites and tails.

Ripple Crossover - 2

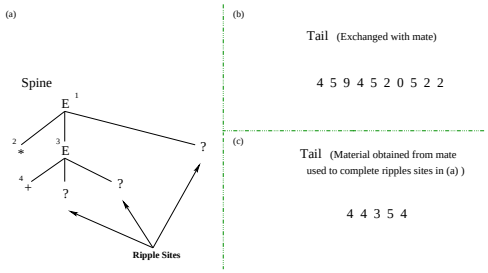


Figure 3: Illustrated are the spine and the resulting ripple sites (a) and tails (b)(c) produced as a consequence of the one-point crossover in Figure 2

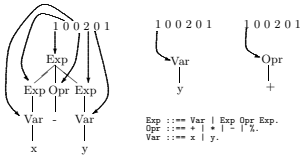


Figure 4: Intrinsic polymorphism: the same string of numbers can decode to different choices, depending on the symbol that they are being grafted onto.

Ripple Crossover - 3

- **Symbolic Regression Grammars**

```
E ::= x
    | (+ E E) | (* E E)
    | (- E E) | (/ E E)
```

And the context free grammar:

```
Exp ::= Var | Exp Op Exp
Var ::= x
Op  ::= + | * | - | /
```

- **Santa Fe ant trail grammars**

```
E ::= move() | left() | right()
    | iffoodahead(E E) | prog2(E, E)
```

And the context free grammar:

```
Code      ::= Line | prog2(Line, Code)
Line      ::= Condition | Action
Action    ::= move() | right() | left()
Condition ::= iffoodahead(Code, Code)
```


Ripple Crossover - 4

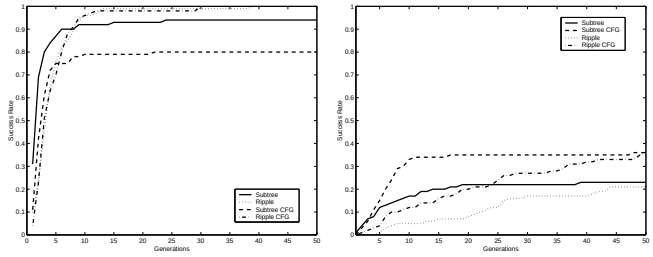


Figure 5: Success rates on the symbolic regression and Santa Fe ant trail problems, averaged over 100 runs.

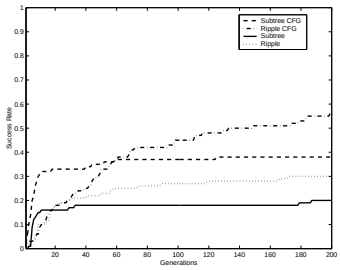


Figure 6: Success rates on the Santa Fe trail problem, averaged over 100 runs, each running for 200 generations.

Other types of Crossover?

- **Homologous Crossover**
  - **Try not to cross in identical areas**
- **Two point**
- **Uniform**
- **Same size homologous**
- **Same size two point**

Homologous Crossover - 1

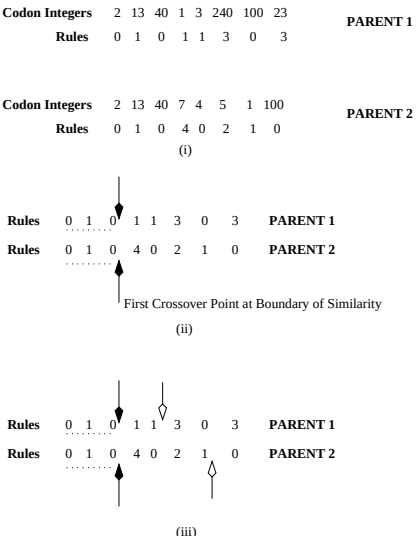


Figure 7: Depicted is standard GE homologous crossover. (i) Shows two parents represented as their codon integer values on top, and the corresponding rules selected during the mapping process below each integer value. (ii) The rule strings (mapping histories) are aligned, and the region of similarity noted (underlined). The first crossover points are selected at this boundary. (iii) The second crossover points are then selected after the boundary of similarity for each individual.

Homologous Crossover - 2

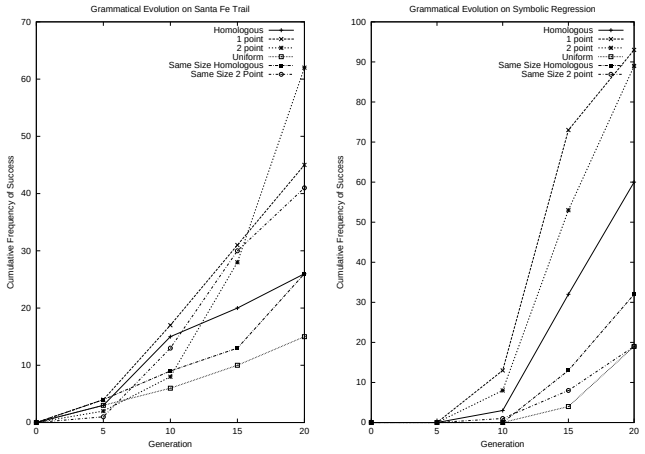


Figure 8: Comparison of the cumulative frequencies of success for each crossover operator on the Santa Fe ant trail and Symbolic Regression problems.

- **1pt/2pt best**
- **Uniform worst**

Homologous Crossover - 2

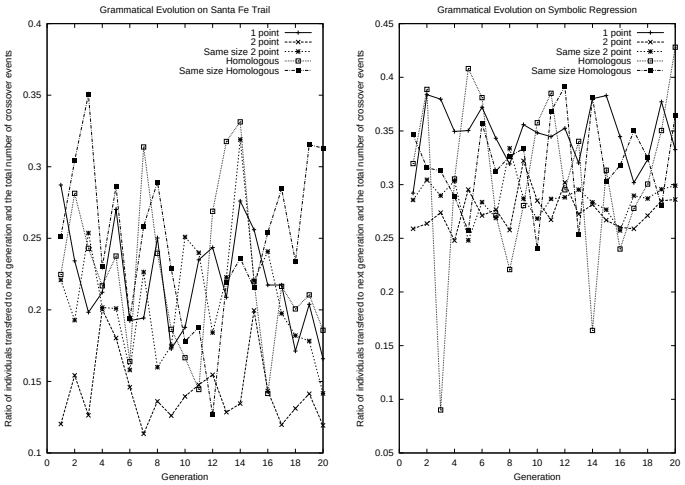


Figure 9: Ratio of the number of individuals undergoing crossover that have been propagated to the next generation and the total number of crossover events occurring in that generation averaged over 20 runs.

Homologous Crossover - 3

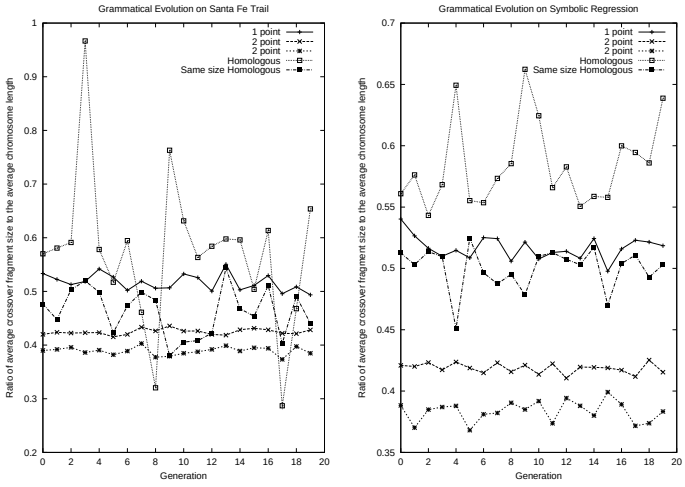


Figure 10: Ratio of the average fragment size being swapped and the average chromosome length at each generation averaged over 20 runs.

- **Appears Crossover works**
- **50% material exchange with 1-point over entire runs**

Headless Chicken

- **If useful material exchanged then swapping random fragments should degrade performance?**

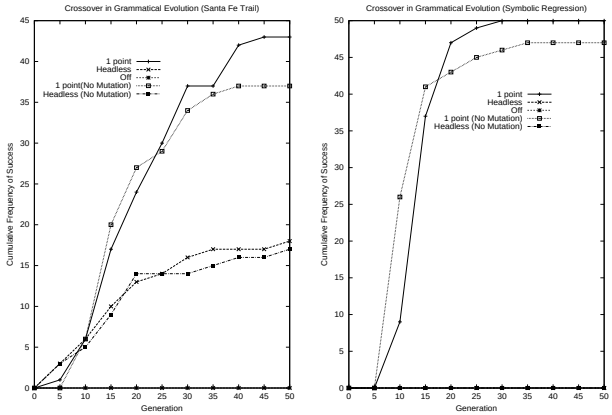


Figure 11: A comparison of GE's performance on the Santa Fe ant trail can be seen on the left. The graph clearly demonstrates the damaging effects of the headless chicken crossover, and in the case when crossover is switched off. A comparison of GE's performance on the symbolic regression problem can be seen on the right. When the headless chicken crossover is used the system fails to find solutions, as is also the case when crossover is switched off.

- **It does!**

Wrapping - 1

- **Wrap Count & Invalid Individuals**

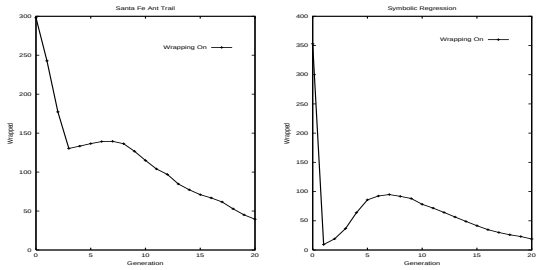


Figure 12: Number of individuals wrapped on the symbolic regression and Santa Fe trail problems.

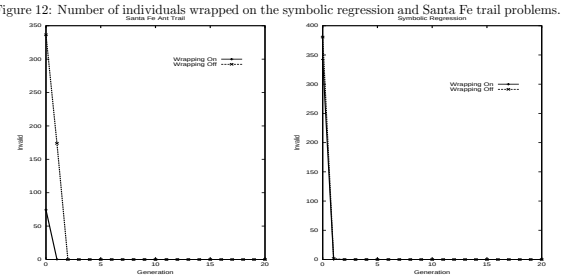


Figure 13: The number of invalid individuals for each generation in the presence and absence of wrapping.

Wrapping - 2

- **Freq. of Success**

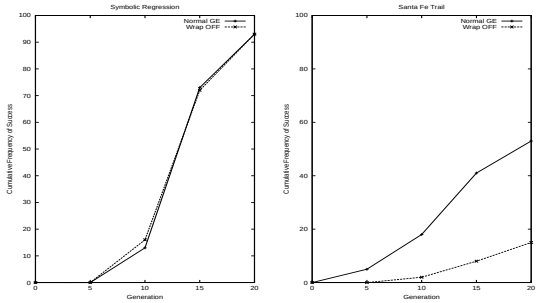


Figure 14: Figure shows the cumulative frequency of success measures on both problems with and without the presence of wrapping.

- **Actual length**
  - **Entire length of individual**
- **Effective length**
  - **Number of codons used**
  - **(Note! Can be less than or greater than actual length)**

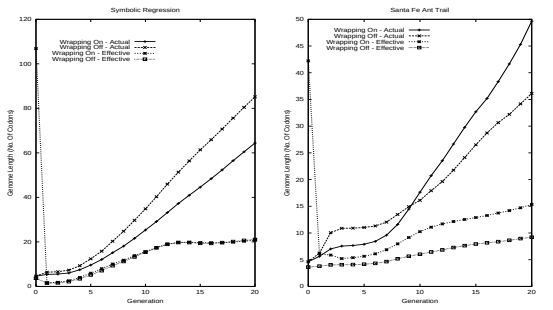


Figure 15: The figure shows the actual versus effective genome length for symbolic regression and the Santa Fe trail in the presence and absence of wrapping.

- **For SR (left) wrapping off has the longest actual length**
- **Effective length virtually the same**
- **For SF (right) wrapping on longer in both cases.**

Wrapping - 3

- **Conclusions:**
  - **Wrapping improves frequency of success on Santa Fe ant trail**
  - **No effect on Symbolic Regression cumulative frequency**
  - **Provides some constraint on genome lengths**

Avoiding fruitless wraps

Single Non-Terminal Grammars

- Only one non-terminal symbol;
- Always same number of choices for each codon.

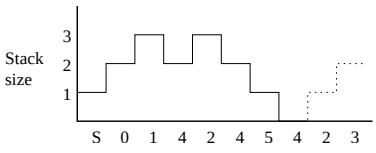
```
<E> ::= ( + <E> <E> )     (0)
      | ( - <E> <E> )     (1)
      | ( * <E> <E> )     (2)
      | ( / <E> <E> )     (3)
      | x                 (4)
      | 1                 (5)
```

- Productions are either Producers, Consumers or Neutrals, and affect the number of remaining non-terminal symbols differently:

Production	PCN number
0	+1
1	+1
2	+1
3	+1
4	-1
5	-1

- Example individual:

Codon	0	1	4	2	4	5	4	2	3
PCN#	1	1	-1	1	-1	-1	-1	1	1



Shape graph

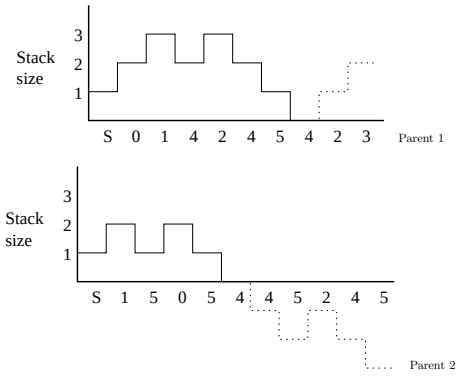
- PCN number for string s is:

$$\sum_{i=1}^{N} pcn(s_i) \quad (1)$$

- Terminating condition:

$$\exists k \in [1..N] : \sum_{i=1}^{k} pcn(s_i) = -1$$

- Stop sequences:
  - Sequence of codons that, on average, tends to consume non-terminals;
  - Steeper slope in shape graph indicates better stop sequences.



Dual Non-Terminal Grammars

- Two non-terminal symbols in grammar:

```
<A> ::= a<A><A>
      | <B>
<B> ::= a
      | b
```

- If Individual does not map after first pass:
  - if top of stack is <A> stop mapping process;
  - if top of stack is <B> wrap individual.
- If Individual does not map after second pass:
  - if top of stack is <A> stop mapping process;
  - if top of stack is <B> and stack size diminished, wrap individual.

General Case

- Difficult to generalize previous method for Multiple Non-Terminal Grammars:

```
<A> ::= <A>
      | <B><C>
<B> ::= b
      | b
<C> ::= <B><B><B>
      | <B><B><B>
```

- Example with individual `01`:

Pass #	Stack status
1	<B><C>
2	b<B><B><B>
3	bbb<B>
4	bbbb

- Same symbol on top of stack at pass 2, but mapping can be completed through wrapping.

- Following heuristic is proposed: If the entire stack from the last pass is at the top of the stack from this pass, then stop mapping process.
- Not guaranteed to stop all fruitless wraps, but guaranteed not to stop mapping process too early.

Wrapping - Experiments

- Purpose:
  - Measure the effectiveness of the heuristic proposed, by comparing the total number of non-mapping individuals with those identified by the heuristic.

Wrapping problem

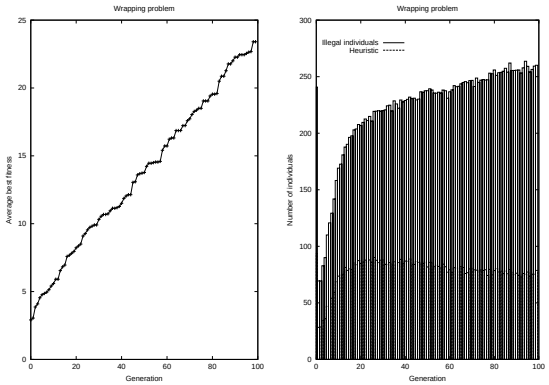
- Fitness = number of wraps required to map (higher is better);

- Grammar:

```
<A> ::= <B><B>| a
<B> ::= <B><B>| <A><A>| c
```

  - Designed to make it difficult to identify non-mapping individuals.

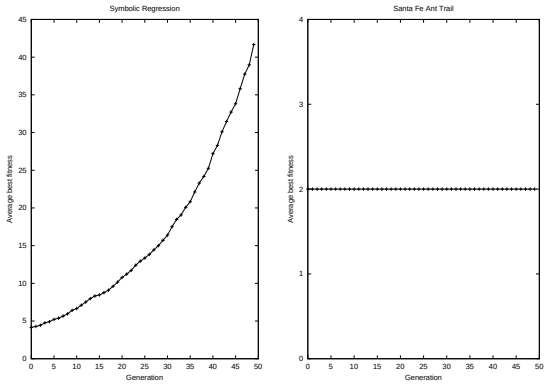
Wrapping problem



Average fitness (left), and comparison of individuals flagged by heuristic versus total non-mapping individuals (right).

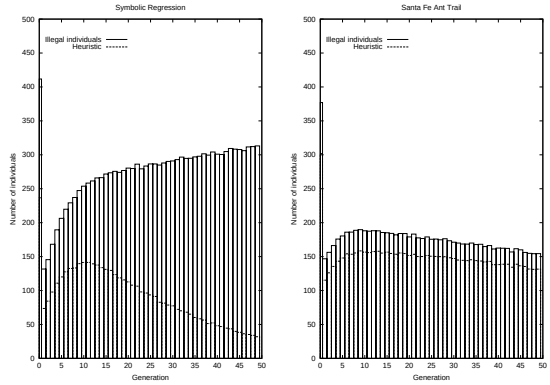
Standard problems

- Symbolic Regression & Santa Fe Ant Trail



Average best fitness for Symbolic Regression and Santa Fe Ant Trail problems.

Heuristic performance:



Comparison of individuals flagged by heuristic versus total non-mapping individuals for Symbolic Regression and Santa Fe Ant Trail problems.

- **Heuristic doesn't eliminate unnecessary wraps completely.**
- **Reduces them even in difficult circumstances**

Degenerate Genetic Code - 1

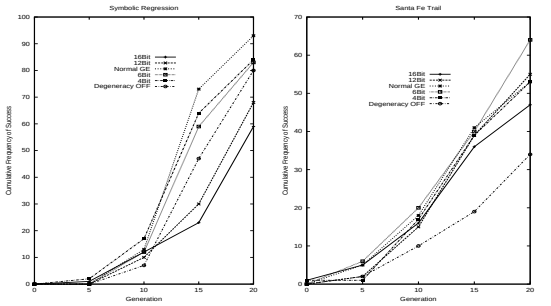


Figure 16: Cumulative frequencies of success for both problem domains in the presence and absence of genetic code degeneracy over 50 generations.

- **No huge difference...**
  - **Normal, 4- and 6-bit top three in both**
  - **No degeneracy fourth in SR, last in SF**

- **Mean variety**

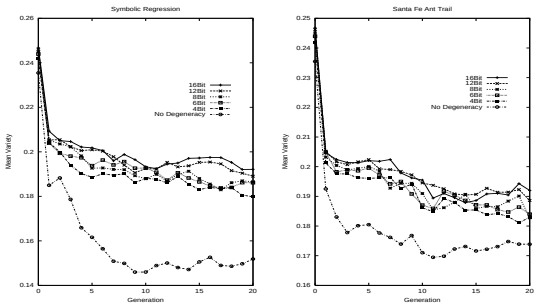


Figure 17: The figure shows the genetic code degeneracy and mean variety on symbolic regression and Santa Fe trail problems.

Degenerate Genetic Code - 2

- **Unique individuals**

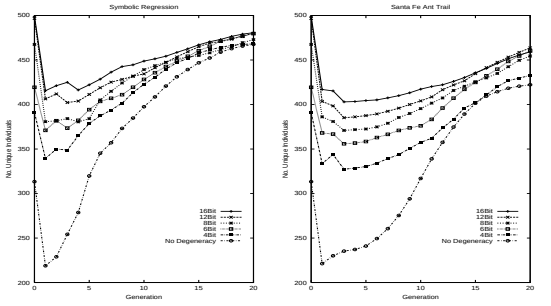


Figure 18: The figure shows genetic code degeneracy and unique individuals (for actual genome) on both problem domains.

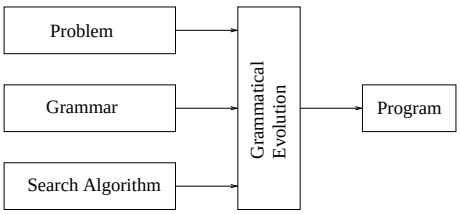
- **Conclusions:**
  - **Improves genetic diversity**
  - **Improves frequency of success on Santa Fe ant trail**
  - **Tunable/Evolvable Degeneracy a good idea?**

Wrapping & Degeneracy

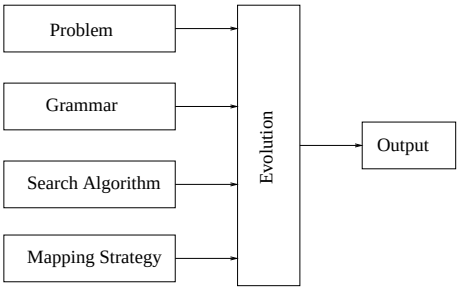
- **Removing both....**
  - **Cumulative frequency of success degrades**
  - **Genome lengths increase over 60% on Symbolic Regression**
  - **Genetic diversity no worse than without degeneracy alone**

Modular Nature

- **Original View :**



- **Current View :**



Chorus

- **Mapping Independent Codons - no ripple effect**
- **Codon %** $Total$ **number of rules in the grammar**
- **Competition between the Genes**
- **Concentration Table**
- **Variable length binary strings**
- **8 bit codons**

```
 S= <expr>
(0) <expr>  ::=   <expr> <op> <expr>
(1)       | ( <expr> <op> <expr> )
(2)       | <pre-op> ( <expr> )
(3)       | <var>
(4) <op>    ::=  +
(5)       | -
(6)       | *
(7)       | /
(8) <pre-op>::= Sin
(9)       | Cos
(A)       | Exp
(B)       | Log
(C) <var>   ::= 1.0
(D)       | X
```


Four non-terminals:

- **<expr> 0..3**
- **<op> 4..7**
- **<pre-op> 8..B**
- **<var> C..D**

```
209 102 190 55 65 15 255 87
D   4   8   D  9  1  3   3
```

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**
<e>	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
<e><o><e>	**0**	*0*	**0**	**0**	**1**	**0**	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**2**
<v><o><e>	**0**	**0**	**0**	*0*	**1**	**0**	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**2**
X<o><e>	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**1**	**1**	**0**	**0**	**0**	*1*
X+<e>	**0**	**0**	**0**	**0**	*0*	**0**	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**1**
X+<v>	**0**	**0**	**0**	*0*	**0**	**0**	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**1**
X+X	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**0**	**0**	**0**	*0*

Advantages:

- No ripple effect (advantage?)
- Significantly better than standard GE on symbolic regression type problems
- Sections of the grammar can be removed from the population
  - "Dependant genes"
  - e.g. all `<pre-op>` rules depend on rule #2

GAuGE - 1

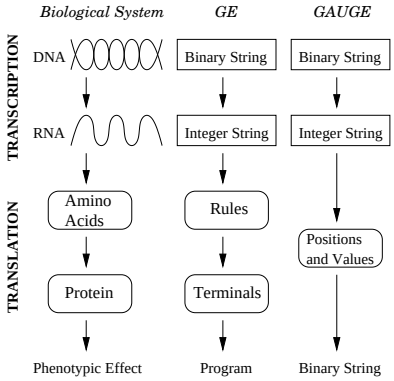
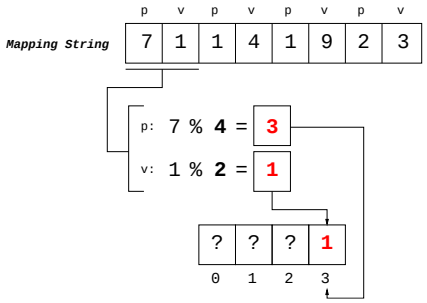
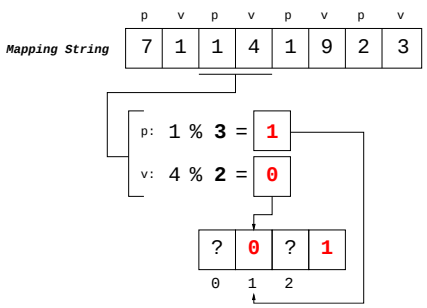


Figure 19:

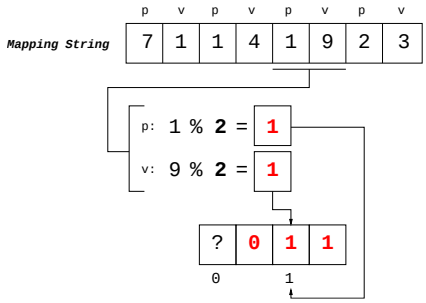
- Start with binary string
- Perform GE-style mapping.
- Produce binary string
- Position independent GA
- Evolution chooses where each gene goes
- Genes at the start are less likely get disrupted

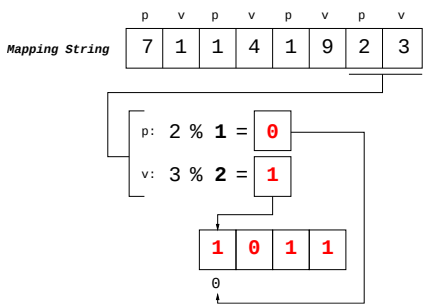
GAuGE - 2





GAuGE - 3





Search Techniques

- **Other techniques**
  - **Simulated Annealing**
  - **Hill Climbing**
  - **Random Search**
- **Three standard GP problems**
  - **Santa Fe trail**
  - **Symbolic Integration (integrate Cos(x) + 2x + 1)**
  - **Symbolic regression** $x^4 + x^3 + x^2 + x$

	Meta heuristic			
Problem	**RS**	**HC**	**SA**	**GA**
Santa Fe	**54%**	**7%**	**14%**	**81%**
Symbolic Integration	**66%**	**4%**	**3%**	**100%**
Symbolic Regression	**0%**	**0%**	**0%**	**59%**

Benchmarks

- **Santa Fe ant trail**
- **Symbolic Regression**
- **Symbolic Integration**
- **Caching algorithms**

Santa Fe ant trail - 1

$$N = \{code, line, expr, if - statement, op\}$$

$$T = \{left(), right(), move(), food\_ahead(), else, if, \{, \}, (,), ;\}$$

$$S =< code >$$

And P can be represented as:

```
(1) <code> :: =  <line>              (0)
               |<code><line>         (1)

(2) <line> :: =  <if-statement>      (0)
               |<op>                 (1)

(3) <if-statement> :: = if(food_ahead())
                          {<line>}
                        else
                          {<line>}

(4) <op> :: =    left();             (0)
             | right();              (1)
             | move();               (2)
```


Objective :	Find a computer program to control an artificial ant so that it can find all 89 pieces of food located on the Santa Fe Trail.
Terminal Operators:	left(), right(), move(), food_ahead()
Terminal Operands:	None
Fitness cases	One fitness case
Raw Fitness	Number of pieces of food before the ant times out with 615 operations.
Standardised Fitness	Total number of pieces of food less the raw fitness.
Wrapper	None
Parameters	Population Size = 500, Termination when Generations = 51 Prob. Mutation = 0.01, Prob. Crossover = 0.9 Prob. Duplication = 0.01, Steady State

Table 1: Grammatical Evolution Tableau for the Santa Fe Trail

Santa Fe ant trail - 2

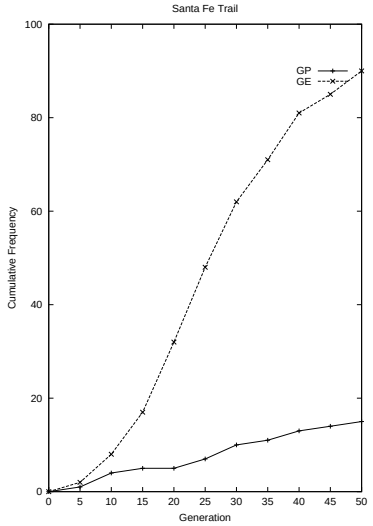


Figure 20: The cumulative frequency of success measure for GE versus GP on the Santa Fe trail problem. The results shown illustrate the case where the solution length constraint is removed from GP.

Symbolic Regression - 1

$$N = \{expr, op, pre\_op\}$$

$$T = \{Sin, Cos, Exp, Log, +, -, /, *, X, 1.0, (,)\}$$

$$S =< expr >$$

And P can be represented as:

```
(1) <expr> ::= <expr> <op> <expr>      (0)
             | ( <expr> <op> <expr> ) (1)
             | <pre-op> ( <expr> )    (2)
             | <var>                  (3)

(2) <op> ::= +       (0)
           | -       (1)
           | /       (2)
           | *       (3)

(3) <pre-op> ::= Sin    (0)
               | Cos    (1)
               | Exp    (2)
               | Log    (3)

(4) <var> ::= X      (0)
            | 1.0    (1)
```


Objective :	Find a function of one independent variable and one dependent variable, in symbolic form that fits a given sample of 20 (x_i, y_i) data points, where the target function is the quartic polynomial $X^4 + X^3 + X^2 + X$
Terminal Operands:	X (the independent variable), 1.0
Terminal Operators	The binary operators $+, *, /$, and $-$ The unary operators Sin, Cos, Exp and Log
Fitness cases	A sample of 20 data points in the interval $[-1, +1]$
Raw Fitness	The sum, taken over the 20 fitness cases of the error
Standardised Fitness	Same as raw fitness
Wrapper	Standard productions to generate C functions
Parameters	Population Size = 500, Termination when Generations = 51 Prob. Mutation = 0.01, Prob. Crossover = 0.9 Prob. Duplication = 0.01, Steady State

Table 2: Symbolic Regression Tableau for GE

Symbolic Regression - 2

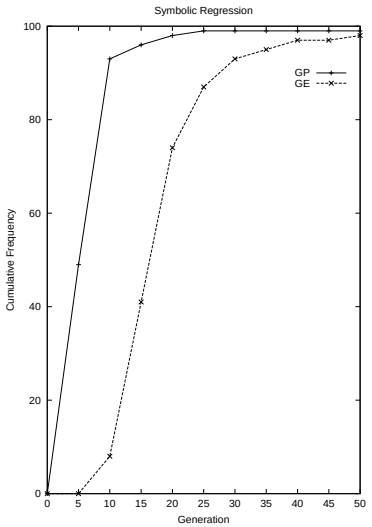


Figure 21: Cumulative frequency of success measure of GE versus GP on the symbolic regression problem.

Symbolic Integration - 1

$$N = \{expr, op, pre\_op\}$$

$$T = \{Sin, Cos, Exp, Log, +, -, /, *, X, 1.0, (,)\}$$

$$S =< expr >$$

And P can be represented as:

```
(1) <expr> ::= <expr> <op> <expr>     (0)
             | ( <expr> <op> <expr> ) (1)
             | <pre-op> ( <expr> )    (2)
             | <var>                  (3)

(2) <op> ::= +       (0)
           | -       (1)
           | /       (2)
           | *       (3)

(3) <pre-op> ::= Sin    (0)
               | Cos    (1)
               | Exp    (2)
               | Log    (3)

(4) <var> ::= X      (0)
            | 1.0    (1)
```


Objective :	Find a function ($f(X) = Sin(X) + X + X^2$), in symbolic form, that is the integral of a curve ($f(X) = Cos(X) + 2X + 1$) presented either as a mathematical expression or as a given finite sample of points (x_i, y_i)
Terminal Operands:	X (the independent variable)
Terminal Operators	The binary operators $+, *, /, -$ and the unary operators Sin, Cos, Exp and Log
Fitness cases	A sample of 20 data points in the interval $[0, 2\pi]$
Raw Fitness	The sum, taken over the 20 fitness cases, of the absolute value of the difference between the individual genetically produced function $f_j(x_i)$ at the domain point x_i and the value of the numerical integral $\mathbf{I}(x_i)$
Standardised Fitness	Same as raw fitness
Wrapper	Standard productions to generate C functions
Parameters	Population Size = 500, Termination when Generations = 51 Prob. Mutation = 0.01, Prob. Crossover = 0.9 Prob. Duplication = 0.01, Steady State

Table 3: Symbolic Integration Tableau for GE

Symbolic Integration - 2

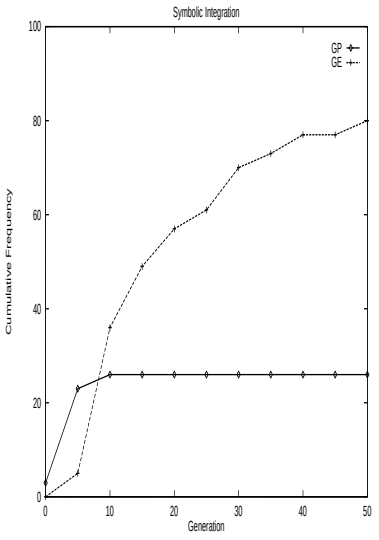


Figure 22: Cumulative frequency of success measure of GE versus GP on the symbolic integration problem.

Caching Algorithms - 1

Terminal Operator	Function
$write\_x(i, v)$:	sets info[i] to v
$read\_x(i)$:	returns info[i]
$small\_x(i, v)$:	index of smallest element of info[]
$large\_x(i, v)$:	index of largest element of info[]
$random\_x(i, v)$:	index of random element of info[]
$counter()$:	successive values 0, 1, 2 etc
$div(x, y)$:	if y==0 then 1 else x / y
$rem(x, y)$:	if y==0 then 1 else x % y

Table 4: Available terminal operators.

```
(1)  <stmts> :: = <stmt>
                | <stmt>;<stmts>

(2)  <stmt> :: = if(<expr>){<stmts>;}else{<stmts>;}
               | write_x(<expr>,<expr>);
               | victim=<expr>;

(3)  <expr> :: = <term>
               | <term>+<term>
               | <term>-<term>
               | <term>*<term>
               | div(<term>,<term>)
               | rem(<term>,<term>)

(4)  <term> :: = CACHESIZE
               | <num>
               | <fun>
               | (<expr>)

(5)  <num> :: = <mant>
              | <mant><zeros>

(6)  <mant> :: = 0 | 1 | 2 | 5

(7)  <zeros> :: = 0
                | 0<zeros>
```

Caching Algorithms - 2

```
(8)  <fun>  :: = counter()
               | read_x(<expr>)
               | small_x()
               | large_x()
               | random_x()
```

```
GE 1 :        victim = counter() - CACHESIZE;
```

Force Use of info[]:

```
(1)  <stmts> :: = write_x(<expr>,<expr>); victim=<expr>;
```

```
GE 2 :   write_x( CACHESIZE + counter(), CACHESIZE + counter())
         victim = CACHESIZE + counter();
```

Algorithm (Cachesize)	**ken2.00100 (Misses)**	**ken2.00200 (Misses)**	**Average of % Improvement over LRU**
LRU (20)	374,596	380,041	-
LRU (200)	367,104	373,935	-
GE1 (20)	300,569	318,444	17.97
GE1 (200)	106,067	82,856	74.51
GE2 (20)	300,569	318,445	17.97
GE2 (200)	106,068	82,855	74.51

Table 5: Algorithm performance comparison.