Parallel Problem Solving from Nature
September 13-17, 2014
Ljubljana, Slovenia
Automatic Design of Algorithms via Hyper-heuristic Genetic Programming

John Woodward, Jerry Swan
John.Woodward@cs.stir.ac.uk; Jerry.Swan@cs.stir.ac.uk
CHORDS Research Group, Stirling University
http://www.maths.stir.ac.uk/research/groups/chords/
Conceptual Overview

Combinatorial problem e.g. Travelling Salesman
Exhaustive search -> heuristic?

Genetic Algorithm
heuristic – permutations

Travelling Salesman
Tour
Single tour NOT EXECUTABLE!!!

Genetic Programming
code fragments in for-loops.

Travelling Salesman Instances
TSP algorithm
EXECUTABLE on MANY INSTANCES!!!

Give a man a fish and he will eat for a day.
Teach a man to fish and he will eat for a lifetime.

Scalable? General?
New domains for GP

09/09/2014 John R. Woodward
Program Spectrum

Genetic Programming
{+, -, *, /}
{AND, OR, NOT}

Automatically
designed heuristics
(this tutorial)

First year university course
On Java, as part of a computer
Science degree

LARGE
Software
Engineering
Projects

Increasing “complexity”
# Overview of Applications

<table>
<thead>
<tr>
<th></th>
<th>SELECTION</th>
<th>MUTATION GA</th>
<th>BIN PACKING</th>
<th>MUTATION EP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalable performance</td>
<td>?</td>
<td>?</td>
<td>Yes - why</td>
<td>No - why</td>
</tr>
<tr>
<td>Generation ZERO</td>
<td>Rank, fitness proportional</td>
<td>NO – needed to seed.</td>
<td>Best fit</td>
<td>Gaussian and Cauchy</td>
</tr>
<tr>
<td>Problem class</td>
<td>Parameterized function</td>
<td>Item size</td>
<td>Parameterized function</td>
<td></td>
</tr>
<tr>
<td>Results Human Competitive</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Algorithm iterate over</td>
<td>Population</td>
<td>Bit-string</td>
<td>Bins</td>
<td>Vector</td>
</tr>
<tr>
<td>Search Method</td>
<td>Random Search</td>
<td>Iterative Hill-Climber</td>
<td>Genetic Programming</td>
<td>Genetic Programming</td>
</tr>
<tr>
<td>Type Signatures</td>
<td>R^2-&gt;R</td>
<td>B^n-&gt;B^n</td>
<td>R^3-&gt;R</td>
<td>()-&gt;R</td>
</tr>
<tr>
<td>Reference</td>
<td>[16]</td>
<td>[15]</td>
<td>[6,9,10,11]</td>
<td>[18]</td>
</tr>
</tbody>
</table>
Plan: From Evolution to Automatic Design

1. (assume knowledge of Evolutionary Computation)
2. Evolution, Genetic Algorithms and Genetic Programming (1)
3. Motivations (conceptual and theoretical) (4)
4. Examples of automatic generation
   Genetic Algorithms (selection and mutation) (8)
   Bin packing (9)
   Evolutionary Programming (12)
8. Wrap up. Closing comments (6)
9. Questions (during AND after...) Please 😊
   Now is a good time to say you are in the wrong room 😊
Evolution GA/GP

- Generate and test: cars, code, models, proofs, medicine, hypothesis.
- Evolution (select, vary, inherit).
- Fit for purpose

Feedback loop
Humans
Computers

Generate

Test

Inheritance
Off-spring have similar Genotype (phenotype) PERFECT CODE [3]

09/09/2014 John R. Woodward
A search space contains the set of all possible solutions.

An objective function determines the quality of solution.

A (Mathematical idealized) metaheuristic determines the sampling order (i.e. enumerates i.e. without replacement). It is a (approximate) permutation. What are we learning?

Performance measure $P(a, f)$ depend only on $y_1, y_2, y_3$

Aim: find a solution with a near-optimal objective value using a Metaheuristic.

ANY QUESTIONS BEFORE NEXT SLIDE?
**Theoretical Motivation 2**

- **Metaheuristic** $a$
- **Search space**
- **permutation** $\sigma$
- **$\sigma^{-1}$**
- **Objective Function** $f$

\[
P(a, f) = P(a \sigma, \sigma^{-1} f) \quad P(A, F) = P(A\sigma, \sigma^{-1} F) \quad \text{(i.e. permute bins)}
\]

$P$ is a **performance measure**, (based only on output values).

$\sigma, \sigma^{-1}$ are a permutation and inverse permutation.

$A$ and $F$ are probability distributions over algorithms and functions).

**F is a problem class. ASSUMPTIONS IMPLICATIONS**

1. **Metaheuristic** $a$ applied to function $\sigma\sigma^{-1} f$ (that is $f$)
2. **Metaheuristic** $a\sigma$ applied to function $\sigma^{-1} f$ precisely identical.
Theoretical Motivation 3 [1,14]

• The base-level learns about the function.
• The meta-level learn about the distribution of functions
• The sets do not need to be finite (with infinite sets, a uniform distribution is not possible)
• The functions do not need to be computable.
• We can make claims about the Kolmogorov Complexity of the functions and search algorithms.
• \( p(f) \) (the probability of sampling a function) is all we can learn in a black-box approach.
One Man – One/Many Algorithm

1. Researchers design heuristics by hand and test them on problem instances or arbitrary benchmarks off internet.

2. Presenting results at conferences and publishing in journals. In this talk/paper we propose a new algorithm...

1. Challenge is defining an algorithmic framework (set) that includes useful algorithms. Black art
2. Let Genetic Programming select the best algorithm for the problem class at hand. Context!!! Let the data speak for itself without imposing our assumptions.

In this talk/paper we propose a 10,000 algorithms...

09/09/2014
John R. Woodward
Evolving Selection Heuristics [16]

• **Rank** selection
  \[ P(i) \propto i \]
  Probability of selection is proportional to the **index** in sorted population

• **Fitness** Proportional
  \[ P(i) \propto \text{fitness}(i) \]
  Probability of selection is proportional to the **fitness**

  *Fitter individuals are more likely to be selected in both cases.*
Framework for Selection Heuristics

Selection heuristics operate in the following framework for all individuals \( p \) in population:

\[
\text{select } p \text{ in proportion to } \text{value}( p );
\]

- To perform rank selection, replace value with index \( i \).
- To perform fitness proportional selection, replace value with fitness.

Space of Programs.

- rank selection is the program.
- fitness proportional
- These are just two programs in our search space.
Selection Heuristic Evaluation

- Selection heuristics are generated by random search in the top layer.
- Heuristics are used as for selection in a GA on a bit-string problem class.
- A value is passed to the upper layer informing it of how well the function performed as a selection heuristic.
Experiments for Selection

• Train on 50 problem instances (i.e. we run a single selection heuristic for 50 runs of a genetic algorithm on a problem instance from our problem class).

• The training times are ignored
  – we are not comparing our generation method.
  – we are comparing our selection heuristic with rank and fitness proportional selection.

• Selection heuristics are tested on a second set of 50 problem instances drawn from the same problem class.
Problem Classes

1. A problem class is a probability distribution of problem instances.
2. Generate values $N(0,1)$ in interval $[-1,1]$ (if we fall outside this range we regenerate)
3. Interpolate values in range $[0, 2^{\text{num-bits}}-1]$
4. Target bit string given by Gray coding of interpolated value.

The above 3 steps generate a distribution of target bit strings which are used for hamming distance problem instances. “shifted ones-max”
Results for Selection Heuristics

<table>
<thead>
<tr>
<th></th>
<th>Fitness Proportional</th>
<th>Rank</th>
<th>generated-selector</th>
</tr>
</thead>
<tbody>
<tr>
<td>mean</td>
<td>0.831528</td>
<td>0.907809</td>
<td>0.916088</td>
</tr>
<tr>
<td>std dev</td>
<td>0.003095</td>
<td>0.002517</td>
<td>0.006958</td>
</tr>
<tr>
<td>min</td>
<td>0.824375</td>
<td>0.902813</td>
<td>0.9025</td>
</tr>
<tr>
<td>max</td>
<td>0.838438</td>
<td>0.914688</td>
<td>0.929063</td>
</tr>
</tbody>
</table>

Performing t-test comparisons of fitness-proportional selection and rank selection against generated heuristics resulted in a p-value of better than $10^{-15}$ in both cases. In both of these cases the generated heuristics outperform the standard selection operators (rank and fit-proportional).
Take Home Points

• automatically designing selection heuristics.
• We should design heuristics for problem classes i.e. with a context/niche/setting.
• This approach is human-competitive (and human cooperative).
• Meta-bias is necessary if we are to tackle multiple problem instances.
• Think frameworks not individual algorithms – we don’t want to solve problem instances we want to solve classes (i.e. many instances from the class)!
Meta and Base Learning [15]

1. At the **base** level we are learning about a **specific** function.
2. At the **meta** level we are learning about the probability distribution.
3. We are just doing “**generate and test**” on “**generate and test**”
4. What is being passed with each **blue arrow**?
5. Training/Testing and Validation
Compare Signatures (Input-Output)

<table>
<thead>
<tr>
<th>Genetic Algorithm</th>
<th>Genetic Algorithm FACTORY</th>
</tr>
</thead>
<tbody>
<tr>
<td>• <em>(B^n → R) → B^n</em></td>
<td>• [((B^n → R)] → ((B^n → R) → B^n)</td>
</tr>
</tbody>
</table>

**Input** is an objective function mapping bit-strings of length n to a real-value.

**Output** is a (near optimal) bit-string i.e. the solution to the problem instance

**Input** is a list of functions mapping bit-strings of length n to a real-value (i.e. sample problem instances from the problem class).

**Output** is a (near optimal) mutation operator for a GA i.e. the solution method (algorithm) to the problem class

We are raising the level of generality at which we operate.

09/09/2014

John R. Woodward
Two Examples of Mutation Operators

- **One point mutation** flips **ONE** single bit in the genome (bit-string).
  (1 point to \( n \) point mutation)
- **Uniform mutation** flips **ALL** bits with a **small probability** \( p \). No matter how we vary \( p \), it will never be one point mutation.
- **Lets invent some more!!!**
- 😞 **NO**, lets build a general method (for problem class)
Off-the-Shelf metaheuristic to Tailor-Make mutation operators for Problem Class

Meta-level
Genetic Programming
Iterative Hill Climbing
(mutation operators)

Fitness value
Mutation operator
Genetic Algorithm
Base-level

search space

One Point mutation
Uniform mutation

Two search spaces
Commonly used Mutation operators

novel mutation heuristics
A program is a list of instructions and arguments. A register is set of addressable memory (R0,..,R4). Negative register addresses means *indirection*. A program can only affect IO registers indirectly. Positive (TRUE) negative (FALSE) on output register.

Insert bit-string on IO register, and extract from IO register.
Arithmetic Instructions

These instructions perform arithmetic operations on the registers.

- **Add** $R_i \leftarrow R_j + R_k$
- **Inc** $R_i \leftarrow R_i + 1$
- **Dec** $R_i \leftarrow R_i - 1$
- **Ivt** $R_i \leftarrow -1 \times R_i$
- **Clr** $R_i \leftarrow 0$
- **Rnd** $R_i \leftarrow \text{Random}([-1, +1])$ //mutation rate
- **Set** $R_i \leftarrow \text{value}$
- **Nop** //no operation or identity
Control-Flow Instructions

These instructions control flow (NOT ARITHMETIC). They include branching and iterative imperatives. Note that this set is not Turing Complete!

- **If** if(Ri > Rj) pc = pc + |Rk| why modulus?
- **IfRand** if(Ri < 100 * random[0,+1]) pc = pc + Rj//allows us to build mutation probabilities WHY?
- **Rpt** Repeat |Ri| times next |Rj| instruction
- **Stp** terminate
## Expressing Mutation Operators

<table>
<thead>
<tr>
<th>Line</th>
<th>UNIFORM</th>
<th>ONE POINT MUTATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Rpt, 33, 18</td>
<td>Rpt, 33, 18</td>
</tr>
<tr>
<td>1</td>
<td>Nop</td>
<td>Nop</td>
</tr>
<tr>
<td>2</td>
<td>Nop</td>
<td>Nop</td>
</tr>
<tr>
<td>3</td>
<td>Nop</td>
<td>Nop</td>
</tr>
<tr>
<td>4</td>
<td>Inc, 3</td>
<td>Inc, 3</td>
</tr>
<tr>
<td>5</td>
<td>Nop</td>
<td>Nop</td>
</tr>
<tr>
<td>6</td>
<td>Nop</td>
<td>Nop</td>
</tr>
<tr>
<td>7</td>
<td>Nop</td>
<td>Nop</td>
</tr>
<tr>
<td>8</td>
<td>IfRand, 3, 6</td>
<td>IfRand, 3, 6</td>
</tr>
<tr>
<td>9</td>
<td>Nop</td>
<td>Nop</td>
</tr>
<tr>
<td>10</td>
<td>Nop</td>
<td>Nop</td>
</tr>
<tr>
<td>11</td>
<td>Nop</td>
<td>Nop</td>
</tr>
<tr>
<td>12</td>
<td>Ivlt, −3</td>
<td>Ivlt, −3</td>
</tr>
<tr>
<td>13</td>
<td>Nop</td>
<td>Stp</td>
</tr>
<tr>
<td>14</td>
<td>Nop</td>
<td>Nop</td>
</tr>
<tr>
<td>15</td>
<td>Nop</td>
<td>Nop</td>
</tr>
<tr>
<td>16</td>
<td>Nop</td>
<td>Nop</td>
</tr>
</tbody>
</table>

- **Uniform mutation**
  - Flips all bits with a fixed probability.
  - **4 instructions**

- **One point mutation**
  - Flips a single bit.
  - **6 instructions**

*Why insert NOP?*

We let GP start with these programs and mutate them.
# 7 Problem Instances

- Problem instances are drawn from a problem class.
- 7 real–valued functions, we will convert to discrete binary optimisations problems for a GA.

<table>
<thead>
<tr>
<th>number</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>2</td>
<td>sin2(x/4 – 16)</td>
</tr>
<tr>
<td>3</td>
<td>(x – 4) * (x – 12)</td>
</tr>
<tr>
<td>4</td>
<td>(x * x – 10 * cos(x))</td>
</tr>
<tr>
<td>5</td>
<td>sin(pi<em>x/64–4) * cos(pi</em>x/64–12)</td>
</tr>
<tr>
<td>6</td>
<td>sin(pi<em>cos(pi</em>x/64 – 12)/4)</td>
</tr>
<tr>
<td>7</td>
<td>1/(1 + x /64)</td>
</tr>
</tbody>
</table>
Function Optimization Problem Classes

1. To test the method we use binary function classes

2. We generate a Normally-distributed value \( t = -0.7 + 0.5 \cdot N(0, 1) \) in the range \([-1, +1]\).

3. We linearly interpolate the value \( t \) from the range \([-1, +1]\) into an integer in the range \([0, 2^{\text{num\text{-}bits}} - 1]\), and convert this into a bit-string \( t' \).

4. To calculate the fitness of an arbitrary bit-string \( x \), the hamming distance between \( x \) and the target bit-string \( t' \) is calculated (giving a value in the range \([0, \text{numbits}]\)). This value is then fed into one of the 7 functions.
## Results – 32 bit problems

<table>
<thead>
<tr>
<th>Problem classes</th>
<th>Uniform Mutation</th>
<th>One-point mutation</th>
<th>generated-mutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Means and standard deviations</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p1 mean</td>
<td>30.82</td>
<td>30.96</td>
<td>31.11</td>
</tr>
<tr>
<td>p1 std-dev</td>
<td>0.17</td>
<td>0.14</td>
<td>0.16</td>
</tr>
<tr>
<td>p2 mean</td>
<td>951</td>
<td>959.7</td>
<td>984.9</td>
</tr>
<tr>
<td>p2 std-dev</td>
<td>9.3</td>
<td>10.7</td>
<td>10.8</td>
</tr>
<tr>
<td>p3 mean</td>
<td>506.7</td>
<td>512.2</td>
<td>528.9</td>
</tr>
<tr>
<td>p3 std-dev</td>
<td>7.5</td>
<td>6.2</td>
<td>6.4</td>
</tr>
<tr>
<td>p4 mean</td>
<td>945.8</td>
<td>954.9</td>
<td>978</td>
</tr>
<tr>
<td>p4 std-dev</td>
<td>8.1</td>
<td>8.1</td>
<td>7.2</td>
</tr>
<tr>
<td>p5 mean</td>
<td>0.262</td>
<td>0.26</td>
<td>0.298</td>
</tr>
<tr>
<td>p5 std-dev</td>
<td>0.009</td>
<td>0.013</td>
<td>0.012</td>
</tr>
<tr>
<td>p6 mean</td>
<td>0.432</td>
<td>0.434</td>
<td>0.462</td>
</tr>
<tr>
<td>p6 std-dev</td>
<td>0.006</td>
<td>0.006</td>
<td>0.004</td>
</tr>
<tr>
<td>p7 mean</td>
<td>0.889</td>
<td>0.89</td>
<td>0.901</td>
</tr>
<tr>
<td>p7 std-dev</td>
<td>0.002</td>
<td>0.003</td>
<td>0.002</td>
</tr>
</tbody>
</table>
# Results – 64 bit problems

<table>
<thead>
<tr>
<th>Problem classes</th>
<th>Uniform Mutation</th>
<th>One-point mutation</th>
<th>generated-mutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Means and stand dev</td>
<td>p1 mean</td>
<td>55.31</td>
<td>56.08</td>
</tr>
<tr>
<td></td>
<td>p1 std-dev</td>
<td>0.33</td>
<td>0.29</td>
</tr>
<tr>
<td></td>
<td>p2 mean</td>
<td>3064</td>
<td>3141</td>
</tr>
<tr>
<td></td>
<td>p2 std-dev</td>
<td>33</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>p3 mean</td>
<td>2229</td>
<td>2294</td>
</tr>
<tr>
<td></td>
<td>p3 std-dev</td>
<td>31</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>p4 mean</td>
<td>3065</td>
<td>3130</td>
</tr>
<tr>
<td></td>
<td>p4 std-dev</td>
<td>36</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>p5 mean</td>
<td>0.839</td>
<td>0.846</td>
</tr>
<tr>
<td></td>
<td>p5 std-dev</td>
<td>0.012</td>
<td>0.01</td>
</tr>
<tr>
<td></td>
<td>p6 mean</td>
<td>0.643</td>
<td>0.643</td>
</tr>
<tr>
<td></td>
<td>p6 std-dev</td>
<td>0.004</td>
<td>0.004</td>
</tr>
<tr>
<td></td>
<td>p7 mean</td>
<td>0.752</td>
<td>0.7529</td>
</tr>
<tr>
<td></td>
<td>p7 std-dev</td>
<td>0.0028</td>
<td>0.004</td>
</tr>
</tbody>
</table>
p-values T Test for 32 and 64-bit functions on the 7 problem classes

<table>
<thead>
<tr>
<th>class</th>
<th>32 bit</th>
<th>32 bit</th>
<th>64 bit</th>
<th>64 bit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Uniform</td>
<td>One-point</td>
<td>Uniform</td>
<td>One-point</td>
</tr>
<tr>
<td>p1</td>
<td>1.98E-08</td>
<td>0.0005683</td>
<td>1.64E-19</td>
<td>1.02E-05</td>
</tr>
<tr>
<td>p2</td>
<td>1.21E-18</td>
<td>1.08E-12</td>
<td>1.63E-17</td>
<td>0.00353</td>
</tr>
<tr>
<td>p3</td>
<td>1.57E-17</td>
<td>1.65E-14</td>
<td>3.49E-16</td>
<td>0.00722</td>
</tr>
<tr>
<td>p4</td>
<td>4.74E-23</td>
<td>1.22E-16</td>
<td>2.35E-21</td>
<td>9.01E-13</td>
</tr>
<tr>
<td>p5</td>
<td>9.62E-17</td>
<td>1.67E-15</td>
<td>4.80E-09</td>
<td>4.23E-06</td>
</tr>
<tr>
<td>p6</td>
<td>2.54E-27</td>
<td>4.14E-24</td>
<td>3.31E-24</td>
<td>3.64E-28</td>
</tr>
<tr>
<td>p7</td>
<td>1.34E-24</td>
<td>3.00E-18</td>
<td>1.45E-28</td>
<td>5.14E-23</td>
</tr>
</tbody>
</table>
Rebuttal to Reviews

1. Did we test the new mutation operators against standard operators (one-point and uniform mutation) on different problem classes?
   • NO – the mutation operator is designed (evolved) specifically for that class of problem.

2. Are we taking the training stage into account?
   • NO, we are just comparing mutation operators in the testing phase – Anyway how could we meaningfully compare “brain power” (manual design) against “processor power” (evolution).

3. Train for all functions – NO, we are specializing.
Additions to Genetic Programming

1. final program is part human constrained part (for-loop) machine generated (body of for-loop).

2. In GP the initial population is typically randomly created. Here we (can) initialize the population with already known good solutions (which also confirms that we can express the solutions). (improving rather than evolving from scratch) – standing on shoulders of giants. Like genetically modified crops – we start from existing crops.

3. Evolving on problem classes (samples of problem instances drawn from a problem class) not instances.

NOW OVER TO JERRY SWAN
Problem Classes Do Occur

1. Problem classes are probability distributions over problem instances.

2. **Travelling Salesman**
   1. Distribution of cities over different counties
   2. E.g. USA is square, Japan is long and narrow.

3. **Bin Packing & Knapsack Problem**
   1. The items are drawn from some probability distribution.

4. Problem classes do occur in the real-world

5. Next slides demonstrate **problem classes** and **scalability** with on-line bin packing.
On-line Bin Packing Problem [9,11]

• A *sequence* of items packed into as few a bins as possible.
• Bin size is 150 units, items uniformly distributed between 20-100.
• Different to the off-line bin packing problem where the set of items.
• The “best fit” heuristic, places the current item in the space it fits best (leaving least slack).
• It has the property that this heuristic does not open a new bin unless it is forced to.

Array of bins

Items packed so far

150 = Bin capacity

Range of Item size 20-100

Sequence of pieces to be packed
Genetic Programming applied to on-line bin packing

Not obvious how to link Genetic Programming to combinatorial problems. The GP tree is applied to each bin with the current item and placed in the bin with the maximum score.

Terminals supplied to Genetic Programming
Initial representation \{C, F, S\}
Replaced with \{E, S\}, E = C - F

John R. Woodward
How the heuristics are applied (skip)
The Best Fit Heuristic

Best fit = 1/(E-S). Point out features.

Pieces of size S, which fit well into the space remaining E, score well.

Best fit applied produces a set of points on the surface. The bin corresponding to the maximum score is picked.
Our best heuristic.

Similar shape to best fit – but curls up in one corner.
Note that this is rotated, relative to previous slide.
Robustness of Heuristics

- all legal results
- some illegal results
Testing Heuristics on problems of much larger size than in training

<table>
<thead>
<tr>
<th>Table I</th>
<th>$H_{trained 100}$</th>
<th>$H_{trained 250}$</th>
<th>$H_{trained 500}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.427768358</td>
<td>0.298749035</td>
<td>0.140986023</td>
</tr>
<tr>
<td>1000</td>
<td>0.406790534</td>
<td>0.010006408</td>
<td>0.000350265</td>
</tr>
<tr>
<td>10000</td>
<td>0.454063071</td>
<td>2.58E-07</td>
<td>9.65E-12</td>
</tr>
<tr>
<td>100000</td>
<td>0.271828318</td>
<td>1.38E-25</td>
<td>2.78E-32</td>
</tr>
</tbody>
</table>

Table shows p-values using the best fit heuristic, for heuristics trained on different size problems, when applied to different sized problems

1. As number of items trained on increases, the probability decreases (see next slide).
2. As the number of items packed increases, the probability decreases (see next slide).
Compared with Best Fit

- Averaged over 30 heuristics over 20 problem instances
- Performance does not deteriorate
- The larger the training problem size, the better the bins are packed.
Compared with Best Fit

- The heuristic seems to learn the number of pieces in the problem
- Analogy with sprinters running a race – accelerate towards end of race.
- The “break even point” is approximately half of the size of the training problem size
- If there is a gap of size 30 and a piece of size 20, it would be better to wait for a better piece to come along later – about 10 items (similar effect at upper bound?).
Designing Mutation Operators for Evolutionary Programming [18]

1. **Evolutionary programming** optimizes functions by evolving a population of real-valued vectors (genotype).

2. **Variation** has been provided (manually) by **probability distributions** (**Gaussian, Cauchy, Levy**).

3. We are **automatically generating** probability distributions (using genetic programming).

4. **Not from scratch**, but from already well known distributions (**Gaussian, Cauchy, Levy**). We are “genetically improving probability distributions”.

5. We are evolving mutation operators **for a problem class** (a probability distributions over functions).

6. **NO CROSSOVER**

Genotype is 
(1.3,...,4.5,...,8.7)
Before mutation

Genotype is 
(1.2,...,4.4,...,8.6)
After mutation
(Fast) Evolutionary Programming

Heart of algorithm is mutation
SO LETS AUTOMATICALLY DESIGN

\[ x_i'(j) = x_i(j) + \eta_i(j)D_j \]

1. **EP** mutates with a Gaussian
2. **FEP** mutates with a Cauchy
3. A **generalization** is mutate with a **distribution D**
(generated with genetic programming)

---

1. Generate the initial population of \( \mu \) individuals, and set \( k = 1 \). Each individual is taken as a pair of real-valued vectors, \((x_i, \eta_i)\), \( \forall i \in \{1, \cdots, \mu\} \).

2. Evaluate the fitness score for each individual \((x_i, \eta_i)\), \( \forall i \in \{1, \cdots, \mu\} \), of the population based on the objective function, \( f(x_i) \).

3. Each parent \((x_i, \eta_i), i = 1, \cdots, \mu\), creates a single offspring \((x_i', \eta_i')\) by: for \( j = 1, \cdots, n \),

\[
\begin{align*}
    x_i'(j) &= x_i(j) + \eta_i(j)N(0, 1), \\
    \eta_i'(j) &= \eta_i(j) \exp(\tau'N(0, 1) + \tau N_j(0, 1))
\end{align*}
\]

where \( x_i(j), x_i'(j), \eta_i(j) \) and \( \eta_i'(j) \) denote the \( j \)-th component of the vectors \( x_i, x_i', \eta_i \) and \( \eta_i' \), respectively. \( N(0, 1) \) denotes a normally distributed one-dimensional random number with mean zero and standard deviation one. \( N_j(0, 1) \) indicates that the random number is generated anew for each value of \( j \). The factors \( \tau \) and \( \tau' \) have commonly set to \( (\sqrt{2(2n)^{-1}}) \) and \( (\sqrt{2n})^{-1} \) [9, 8].

4. Calculate the fitness of each offspring \((x_i', \eta_i')\), \( \forall i \in \{1, \cdots, \mu\} \).

5. Conduct pairwise comparison over the union of parents \((x_i, \eta_i)\) and offspring \((x_i', \eta_i')\), \( \forall i \in \{1, \cdots, \mu\} \). For each individual, \( q \) opponents are chosen randomly from all the parents and offspring with an equal probability. For each comparison, if the individual’s fitness is no greater than the opponent’s, it receives a “win.”

6. Select the \( \mu \) individuals out of \((x_i, \eta_i)\) and \((x_i', \eta_i')\), \( \forall i \in \{1, \cdots, \mu\} \), that have the most wins to be parents of the next generation.

7. Stop if the stopping criterion is satisfied; otherwise, \( k = k + 1 \) and go to Step 3.
Optimization & Benchmark Functions

A set of 23 benchmark functions is typically used in the literature. **Minimization** \( \forall x \in S : f(x_{\text{min}}) \leq f(x) \)

We use them as **problem classes**.

Table 1: The 23 test functions used in our experimental studies, where \( n \) is the dimension of the function, \( f_{\text{min}} \) the minimum value of the function, and \( S \subset \mathbb{R}^n \).

<table>
<thead>
<tr>
<th>Test function</th>
<th>( n )</th>
<th>( S )</th>
<th>( f_{\text{min}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f_1(x) = \sum_{i=1}^{n} x_i^2 )</td>
<td>30</td>
<td>([-100, 100]^n)</td>
<td>0</td>
</tr>
<tr>
<td>( f_2(x) = \sum_{i=1}^{n}</td>
<td>x_i</td>
<td>+ \prod_{i=1}^{n}</td>
<td>x_i</td>
</tr>
<tr>
<td>( f_3(x) = \sum_{i=1}^{n} (\sum_{j=1}^{i} x_j)^2 )</td>
<td>30</td>
<td>([-100, 100]^n)</td>
<td>0</td>
</tr>
<tr>
<td>( f_4(x) = \max_i {</td>
<td>x_i</td>
<td>, 1 \leq i \leq n } )</td>
<td>30</td>
</tr>
<tr>
<td>( f_5(x) = \sum_{i=1}^{n-1} [100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2] )</td>
<td>30</td>
<td>([-30, 30]^n)</td>
<td>0</td>
</tr>
<tr>
<td>( f_6(x) = \sum_{i=1}^{n} x_i + 0.5 )</td>
<td>30</td>
<td>([-100, 100]^n)</td>
<td>0</td>
</tr>
<tr>
<td>( f_7(x) = \sum_{i=1}^{n} i x_i^4 + \text{random}[0, 1] )</td>
<td>30</td>
<td>([-1.28, 1.28]^n)</td>
<td>0</td>
</tr>
<tr>
<td>( f_8(x) = \sum_{i=1}^{n} -x_i \sin(\sqrt{</td>
<td>x_i</td>
<td>}) )</td>
<td>30</td>
</tr>
<tr>
<td>( f_9(x) = \sum_{i=1}^{n} x_i^2 - 10 \cos(2\pi x_i) + 10 )</td>
<td>30</td>
<td>([-5.12, 5.12]^n)</td>
<td>0</td>
</tr>
<tr>
<td>( f_{10}(x) = -20 \exp \left( -0.2 \sqrt{\frac{1}{n} \sum_{i=1}^{n} x_i^2} \right) - \exp \left( \frac{1}{n} \sum_{i=1}^{n} \cos 2\pi x_i \right) + 20 + e )</td>
<td>30</td>
<td>([-32, 32]^n)</td>
<td>0</td>
</tr>
</tbody>
</table>
Function Class 1

1. Machine learning needs to generalize.
2. We generalize to function classes.
3. \( y = x^2 \) (a function)
4. \( y = ax^2 \) (parameterised function)
5. \( y = ax^2, \ a \sim [1, 2] \) (function class)
6. We do this for all benchmark functions.
7. The mutation operators is evolved to fit the probability distribution of functions.
## Function Classes 2

<table>
<thead>
<tr>
<th>Function</th>
<th>Domain</th>
<th>Global Minimum</th>
<th>Global Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1(x) = a \sum_{i=1}^{n} x_i^2$</td>
<td>$[-100, 100]^n$</td>
<td>N/A</td>
<td>0</td>
</tr>
<tr>
<td>$f_2(x) = a \sum_{i=1}^{n}</td>
<td>x_i</td>
<td>+ b \prod_{i=1}^{n}</td>
<td>x_i</td>
</tr>
<tr>
<td>$f_3(x) = \sum_{i=1}^{n} (a \sum_{j=1}^{i} x_j)^2$</td>
<td>$[-100, 100]^n$</td>
<td>N/A</td>
<td>0</td>
</tr>
<tr>
<td>$f_4(x) = \max_{i} {a</td>
<td>x_i</td>
<td>, 1 \leq i \leq n}$</td>
<td>$[-100, 100]^n$</td>
</tr>
<tr>
<td>$f_5(x) = \sum_{i=1}^{n} [a(x_{i+1} - x_i)^2 + (x_i - 1)^2]$</td>
<td>$[-30, 30]^n$</td>
<td>N/A</td>
<td>0</td>
</tr>
<tr>
<td>$f_6(x) = \sum_{i=1}^{n} ([ax_i + 0.5])^2$</td>
<td>$[-100, 100]^n$</td>
<td>N/A</td>
<td>0</td>
</tr>
<tr>
<td>$f_7(x) = a \sum_{i=1}^{n} ix_i^4 + \text{random}[0, 1)$</td>
<td>$[-1.28, 1.28]^n$</td>
<td>N/A</td>
<td>0</td>
</tr>
<tr>
<td>$f_8(x) = \sum_{i=1}^{n} -(x_i \sin(\sqrt{</td>
<td>x_i</td>
<td>}) + a)$</td>
<td>$[-500, 500]^n$</td>
</tr>
<tr>
<td>$f_9(x) = \sum_{i=1}^{n} [ax_i^2 + b(1 - \cos(2\pi x_i))]$</td>
<td>$[-5.12, 5.12]^n$</td>
<td>$b \in [5, 10]$</td>
<td>0</td>
</tr>
<tr>
<td>$f_{10}(x) = -a \exp(-0.2 \sqrt{\frac{1}{n} \sum_{i=1}^{n} x_i^2}) - \exp(\frac{1}{n} \sum_{i=1}^{n} \cos 2\pi x_i) + a + e$</td>
<td>$[-32, 32]^n$</td>
<td>N/A</td>
<td>0</td>
</tr>
</tbody>
</table>
Meta and Base Learning

- At the **base** level we are learning about a **specific** function.
- At the **meta** level we are learning about the problem **class**.
- We are just doing **“generate and test”** at a higher level.
- What is being passed with each **blue arrow**?
- **Conventional EP**
Compare Signatures (Input-Output)

<table>
<thead>
<tr>
<th>Evolutionary Programming</th>
<th>Evolutionary Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>((R^n \to R) \to R^n)</td>
<td>([(R^n \to R)] \to ((R^n \to R) \to R^n))</td>
</tr>
</tbody>
</table>

**Input** is a function mapping real-valued vectors of length \(n\) to a real-value.

**Output** is a (near optimal) real-valued vector (i.e. the solution to the problem instance)

**Input** is a list of functions mapping real-valued vectors of length \(n\) to a real-value (i.e. sample problem instances from the problem class).

**Output** is a (near optimal) (mutation operator for) Evolutionary Programming (i.e. the solution method to the problem class)

We are raising the level of generality at which we operate.
Genetic Programming to Generate Probability Distributions

1. GP Function Set \{+, -, *, \%\}
2. GP Terminal Set \{N(0, \text{random})\}

N(0,1) is a normal distribution.

For example a Cauchy distribution is generated by \(N(0,1) \% N(0,1)\).

Hence the search space of probability distributions contains the two existing probability distributions used in EP but also novel probability distributions.
Means and Standard Deviations

These results are good for two reasons.

1. **starting** with a manually designed distributions (Gaussian).
2. evolving distributions **for each function class**.

<table>
<thead>
<tr>
<th>Function Class</th>
<th>FEP Mean</th>
<th>Best</th>
<th>Std Dev</th>
<th>CEP Mean</th>
<th>Best</th>
<th>Std Dev</th>
<th>GP-distribution Mean</th>
<th>Best</th>
<th>Std Dev</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1$</td>
<td>1.24x10^{-3}</td>
<td>2.69x10^{-4}</td>
<td>1.45x10^{-4}</td>
<td>9.95x10^{-5}</td>
<td>6.37x10^{-5}</td>
<td>5.56x10^{-5}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$f_2$</td>
<td>1.53x10^{-1}</td>
<td>2.72x10^{-2}</td>
<td>4.30x10^{-2}</td>
<td>9.08x10^{-3}</td>
<td>8.14x10^{-4}</td>
<td>8.50x10^{-4}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$f_3$</td>
<td>2.74x10^{-2}</td>
<td>2.43x10^{-2}</td>
<td>5.15x10^{-2}</td>
<td>9.52x10^{-2}</td>
<td>6.14x10^{-3}</td>
<td>8.78x10^{-3}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$f_4$</td>
<td>1.79</td>
<td>1.84</td>
<td>1.75x10</td>
<td>6.10</td>
<td>2.16x10^{-1}</td>
<td>6.54x10^{-1}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$f_5$</td>
<td>2.52x10^{-3}</td>
<td>4.96x10^{-4}</td>
<td>2.66x10^{-4}</td>
<td>4.65x10^{-5}</td>
<td>8.39x10^{-7}</td>
<td>1.43x10^{-7}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$f_6$</td>
<td>3.86x10^{-2}</td>
<td>3.12x10^{-2}</td>
<td>4.40x10</td>
<td>1.42x10^2</td>
<td>9.20x10^{-3}</td>
<td>1.34x10^{-2}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$f_7$</td>
<td>6.49x10^{-2}</td>
<td>1.04x10^{-2}</td>
<td>6.64x10^{-2}</td>
<td>1.21x10^{-2}</td>
<td>5.25x10^{-2}</td>
<td>8.46x10^{-3}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$f_8$</td>
<td>-11342.0</td>
<td>3.26x10^2</td>
<td>-7894.6</td>
<td>6.14x10^2</td>
<td>-12611.6</td>
<td>2.30x10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$f_9$</td>
<td>6.24x10^{-2}</td>
<td>1.30x10^{-2}</td>
<td>1.09x10^2</td>
<td>3.58x10</td>
<td>1.74x10^{-3}</td>
<td>4.25x10^{-4}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$f_{10}$</td>
<td>1.67</td>
<td>4.26x10^{-1}</td>
<td>1.45</td>
<td>2.77x10^{-1}</td>
<td>1.38</td>
<td>2.45x10^{-1}</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# T-tests

Table 5 2-tailed t-tests comparing EP with GP-distributions, FEP and CEP on \( f_1-f_{10} \).

<table>
<thead>
<tr>
<th>Function Class</th>
<th>Number of Generations</th>
<th>GP-distribution vs FEP t-test</th>
<th>GP-distribution vs CEP t-test</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f_1 )</td>
<td>1500</td>
<td>( 2.78 \times 10^{-47} )</td>
<td>( 4.07 \times 10^{-2} )</td>
</tr>
<tr>
<td>( f_2 )</td>
<td>2000</td>
<td>( 5.53 \times 10^{-62} )</td>
<td>( 1.59 \times 10^{-54} )</td>
</tr>
<tr>
<td>( f_3 )</td>
<td>5000</td>
<td>( 8.03 \times 10^{-8} )</td>
<td>( 1.14 \times 10^{-3} )</td>
</tr>
<tr>
<td>( f_4 )</td>
<td>5000</td>
<td>( 1.28 \times 10^{-7} )</td>
<td>( 3.73 \times 10^{-36} )</td>
</tr>
<tr>
<td>( f_5 )</td>
<td>200000</td>
<td>( 2.80 \times 10^{-58} )</td>
<td>( 9.29 \times 10^{-63} )</td>
</tr>
<tr>
<td>( f_6 )</td>
<td>1500</td>
<td>( 1.85 \times 10^{-8} )</td>
<td>( 3.11 \times 10^{-2} )</td>
</tr>
<tr>
<td>( f_7 )</td>
<td>3000</td>
<td>( 3.27 \times 10^{-9} )</td>
<td>( 2.00 \times 10^{-9} )</td>
</tr>
<tr>
<td>( f_8 )</td>
<td>9000</td>
<td>( 7.99 \times 10^{-48} )</td>
<td>( 5.82 \times 10^{-75} )</td>
</tr>
<tr>
<td>( f_9 )</td>
<td>5000</td>
<td>( 6.37 \times 10^{-55} )</td>
<td>( 6.54 \times 10^{-39} )</td>
</tr>
<tr>
<td>( f_{10} )</td>
<td>1500</td>
<td>( 9.23 \times 10^{-5} )</td>
<td>( 1.93 \times 10^{-1} )</td>
</tr>
</tbody>
</table>
Table 8: This table compares the fitness values (averaged over 20 runs) of each of the 23 ADRs on each of the 23 function classes. Standard deviations are in parentheses.

<table>
<thead>
<tr>
<th>ADR 1</th>
<th>ADR 2</th>
<th>ADR 3</th>
<th>ADR 4</th>
<th>ADR 5</th>
<th>ADR 6</th>
<th>ADR 7</th>
<th>ADR 8</th>
<th>ADR 9</th>
<th>ADR 10</th>
<th>ADR 11</th>
<th>ADR 12</th>
<th>ADR 13</th>
<th>ADR 14</th>
<th>ADR 15</th>
<th>ADR 16</th>
<th>ADR 17</th>
<th>ADR 18</th>
<th>ADR 19</th>
<th>ADR 20</th>
<th>ADR 21</th>
<th>ADR 22</th>
<th>ADR 23</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.000623453</td>
<td>0.000042932</td>
<td>0.000123910</td>
<td>0.000040239</td>
<td>0.000123910</td>
<td>0.000040239</td>
<td>0.000123910</td>
<td>0.000040239</td>
<td>0.000123910</td>
<td>0.000040239</td>
<td>0.000123910</td>
<td>0.000040239</td>
<td>0.000123910</td>
<td>0.000040239</td>
<td>0.000123910</td>
<td>0.000040239</td>
<td>0.000123910</td>
<td>0.000040239</td>
<td>0.000123910</td>
<td>0.000040239</td>
<td>0.000123910</td>
<td>0.000040239</td>
<td>0.000123910</td>
</tr>
</tbody>
</table>

Performance on Other Problem Classes

09/07/2014
John R. Woodward
Step by Step Guide to Automatic Design of Algorithms [8, 12]

1. Study the literature for existing heuristics for your chosen domain (manually designed heuristics).

2. Build an algorithmic framework or template which expresses the known heuristics (todo ref).

3. Let Genetic Programming supply variations on the theme.

4. Train and test on problem instances drawn from the same probability distribution (like machine learning). Constructing an optimizer is machine learning (this approach prevents “cheating”).
A Brief History (Example Applications) [5]

1. **Image Recognition** – Roberts Mark
2. **Travelling Salesman Problem** – Keller Robert
3. **Boolean Satisfiability** – Fukunaga, Bader-El-Den
4. **Data Mining** – Gisele L. Pappa, Alex A. Freitas
5. **Decision Tree** - Gisele L. Pappa et. al.
6. **Selection Heuristics** – Woodward & Swan
7. **Bin Packing 1,2,3 dimension** (on and off line) 
   Edmund Burke et. al. & Riccardo Poli et. al.
8. **Bug Location** – Shin Yoo
9. **Job Shop Scheduling** - Mengjie Zhang
Comparison of Search Spaces

• If we tackle a problem instance directly, e.g. Travelling Salesman Problem, we get a combinatorial explosion. The search space consists of solutions, and therefore explodes as we tackle larger problems.

• If we tackle a generalization of the problem, we do not get an explosion as the distribution of functions expressed in the search space tends to a limiting distribution. The search space consists of algorithms to produce solutions to a problem instance of any size.

• The algorithm to tackle TSP of size 100-cities, is the same size as the algorithm to tackle TSP of size 10,000-cities.
A Paradigm Shift?

One person proposes one algorithm and tests it in isolation.

One person proposes a family of algorithms and tests them in the context of a problem class.

- Previously one person proposes one algorithm
- Now one person proposes a set of algorithms
- Analogous to “industrial revolution” from hand made to machine made. Automatic Design.

Human cost (INFLATION)  machine cost MOORE’S LAW

conventional approach  new approach
Conclusions

1. Heuristic are trained to fit a problem class, so are designed in context (like evolution). Let’s close the feedback loop! Problem instances live in classes.

2. We can design algorithms on small problem instances and scale them apply them to large problem instances (TSP, child multiplication).
## Overview of Applications

<table>
<thead>
<tr>
<th></th>
<th>SELECTION</th>
<th>MUTATION GA</th>
<th>BIN PACKING</th>
<th>MUTATION EP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalable performance</td>
<td>?</td>
<td>?</td>
<td>Yes - why</td>
<td>No - why</td>
</tr>
<tr>
<td>Generation ZERO</td>
<td>Rank, fitness proportional</td>
<td>NO – needed to seed.</td>
<td>Best fit</td>
<td>Gaussian and Cauchy</td>
</tr>
<tr>
<td>Problem class</td>
<td></td>
<td>Parameterized function</td>
<td>Item size</td>
<td>Parameterized function</td>
</tr>
<tr>
<td>Results Human Competitive</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Algorithm iterate over</td>
<td>Population</td>
<td>Bit-string</td>
<td>Bins</td>
<td>Vector</td>
</tr>
<tr>
<td>Search Method</td>
<td>Random Search</td>
<td>Iterative Hill-Climber</td>
<td>Genetic Programming</td>
<td>Genetic Programming</td>
</tr>
<tr>
<td>Type Signatures</td>
<td>R^2-&gt;R</td>
<td>B^n-&gt;B^n</td>
<td>R^3-&gt;R</td>
<td>()-&gt;R</td>
</tr>
<tr>
<td>Reference</td>
<td>[16]</td>
<td>[15]</td>
<td>[6,9,10,11]</td>
<td>[18]</td>
</tr>
</tbody>
</table>
1. We can automatically design algorithms that **consistently outperform human designed algorithms (on various domains)**.

2. Humans should not provide variations—genetic programing can do that.

3. We are altering the heuristic to suit the set of problem instances presented to it, in the hope that it will generalize to new problem instances (**same distribution - central assumption in machine learning**).

4. The “best” heuristics **depends on the set of problem instances**. (**feedback**)

5. Resulting algorithm is **part man-made part machine-made (synergy)**

6. **not evolving from scratch like Genetic Programming**, improve existing algorithms and adapt them to the new problem instances.

7. Humans are working at a **higher level of abstraction** and more creative. Creating search spaces for GP to sample.

8. **Algorithms are reusable, “solutions” aren’t**. (e.g. tsp algorithm vs route)

9. **Opens up new problem domains**. E.g. bin-packing.
• Thank you for listening
• I am glad to take any
  – comments (+,-)
  – Criticisms
Please email me any references.
http://www.cs.stir.ac.uk/~jrw/
References 1


References


