Hyper-heuristics Tutorial

John Woodward (John.Woodward@cs.stir.ac.uk)
CHORDS Research Group, Stirling University
(http://www.maths.stir.ac.uk/research/groups/chords/)

Daniel R. Tauritz (dtauritz@acm.org)
Natural Computation Laboratory, Missouri University of Science and Technology (http://web.mst.edu/~tauritzd/nc-lab/)

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.

Copyright is held by the owner/author(s).
GECCO’15 Companion, July 11-15, 2015, Madrid, Spain
ACM 978-1-4503-3488-4/15/07.
http://dx.doi.org/10.1145/2739482.2756579

8 May, 2015
John’s perspective of hyperheuristics
Conceptual Overview

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

Genetic Algorithm
heuristic – permutations

Travelling Salesman
Tour

Genetic Programming
code fragments in for-loops.

Travelling Salesman Instances
TSP algorithm

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

EXECUTABLE on MANY INSTANCES!!!

Scalable? General?
New domains for GP
Program Spectrum

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

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

Automatically designed heuristics
(this tutorial)

LARGE Software Engineering Projects

Increasing “complexity”
Plan: From Evolution to Automatic Design

1. Evolution, Genetic Algorithms and Genetic Programming
2. Motivations (conceptual and theoretical)
3. Examples of Automatic Generation:
   • Evolutionary Algorithms (selection, mutation, crossover)
   • Black Box Search Algorithms
   • Bin packing
   • Evolutionary Programming
4. Visualization
5. Step-by-step guide
6. Wrap up (comparison, history, conclusions, summary, etc.)
7. 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
1. A **search space** contains the set of all possible solutions.
2. An **objective function** determines the quality of solution.
3. A **(Mathematical idealized) metaheuristic** determines the sampling order (i.e. enumerates i.e. without replacement). It is an (approximate) permutation. What are we learning?
4. Performance measure $P(a, f)$ depend only on $y_1, y_2, y_3$
5. **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 $f$

$P(a, f) = P(\sigma a, \sigma^{-1} f)$  $P(A, F) = P(A \sigma, \sigma^{-1} F)$ (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...
Daniel’s perspective of hyper-heuristics
Real-World Challenges

• Researchers strive to make algorithms increasingly general-purpose
• But practitioners have very specific needs
• Designing custom algorithms tuned to particular problem instance distributions and/or computational architectures can be very time consuming
Automated Design of Algorithms

• Addresses the need for custom algorithms
• But due to high computational complexity, only feasible for repeated problem solving
• Hyper-heuristics accomplish automated design of algorithms by searching program space
Hyper-heuristics

• Hyper-heuristics are a special type of meta-heuristic
  – Step 1: Extract algorithmic primitives from existing algorithms
  – Step 2: Search the space of programs defined by the extracted primitives

• While Genetic Programming (GP) is particularly well suited for executing Step 2, other meta-heuristics can be, and have been, employed

• The type of GP employed matters [24]
Case Study 1: The Automated Design of Selection Heuristics
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.

Current population (index, fitness, bit-string)

Next generation

1 5.5 0100010 2 7.5 0101010 3 8.9 0001010 4 9.9 0111010

0001010 0111010 0001010 0100010

0001010 0111010 0001010 0100010
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.

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.

Generate a selection heuristic

Genetic Algorithm

bit-string problem

Generate and test X²

Program space of selection heuristics

Framework for selection heuristics. Selection function plugs into a Genetic Algorithm

Problem class:
A probability distribution
Over bit-string problems
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)!
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

---

8 May, 2015

John R. Woodward, Daniel R. Tauritz
# Compare Signatures (Input-Output)

<table>
<thead>
<tr>
<th>Genetic Algorithm</th>
<th>Genetic Algorithm FACTORY</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\bullet \ (B^n \rightarrow R) \rightarrow B^n$</td>
<td>$\bullet \ [{(B^n \rightarrow R)}] \rightarrow {(B^n \rightarrow R) \rightarrow B^n}$</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.
Case Study 2: The Automated Design of Crossover Operators [20]
Motivation

• Performance Sensitive to Crossover Selection

• Identifying & Configuring Best Traditional Crossover is Time Consuming

• Existing Operators May Be Suboptimal

• Optimal Operator May Change During Evolution
Some Possible Solutions

• Meta-EA
  – Exceptionally time consuming

• Self-Adaptive Algorithm Selection
  – Limited by algorithms it can choose from
Self-Configuring Crossover (SCX)

• Each Individual Encodes a Crossover Operator

• Crossovers Encoded as a List of Primitives
  – Swap
  – Merge

• Each Primitive has three parameters
  – Number, Random, or Inline

Offspring Crossover
  Swap(3, 5, 2)
  Merge(1, r, 0.7)
  Swap(r, i, r)
Applying an SCX

Concatenate Genes

Parent 1 Genes

1.0 2.0 3.0 4.0

Parent 2 Genes

5.0 6.0 7.0 8.0
The Swap Primitive

• Each Primitive has a type
  – Swap represents crossovers that move genetic material

• First Two Parameters
  – Start Position
  – End Position

• Third Parameter Primitive Dependent
  – Swaps use “Width”
Applying an SCX

Concatenate Genes

Offspring Crossover

Swap(3, 5, 2)

Merge(1, r, 0.7)

Swap(r, i, r)
The Merge Primitive

• Third Parameter Primitive Dependent
  – Merges use “Weight”

• Random Construct
  – All past primitive parameters used the Number construct
  – “r” marks a primitive using the Random Construct
  – Allows primitives to act stochastically

Merge(1, r, 0.7)
Applying an SCX

Offspring Crossover

Swap(3, 5, 2)

Merge(1, r, 0.7)

Swap(r, i, r)

Concatenate Genes

\[
g(1) = 1.0 \times 0.7 + 6.0 \times (1-0.7)\\
g(i) = \alpha g(i) + (1-\alpha)g(j)\\
g(2) = 6.0 \times 0.7 + 1.0 \times (1-0.7) = 4.5
\]
The Inline Construct

• Only Usable by First Two Parameters

• Denoted as “i”

• Forces Primitive to Act on the Same Loci in Both Parents
Applying an SCX

Offspring Crossover

Swap(3, 5, 2)

Merge(1, r, 0.7)

Swap(r, i, r)

Concatenate Genes

2.5 2.0 5.0 4.5 3.0 4.0 7.0 8.0
Applying an SCX

Remove Exess Genes
Create Genes

Offspring Genes

2.5  4.0  5.0  4.5  3.0  2.0  7.0  8.0
Evolving Crossovers

Parent 1 Crossover
- $\text{Merge}(i, 8, r)$
- $\text{Merge}(1, r, 0.7)$

Parent 2 Crossover

Offspring Crossover
- $\text{Swap}(3, 5, 2)$

Swap ($r, 7, 3$)

Swap ($4, 2, r$)

Merge ($r, r, r$)

Swap ($r, i, r$)
Empirical Quality Assessment

- Compared Against
  - Arithmetic Crossover
  - N-Point Crossover
  - Uniform Crossover

- On Problems
  - Rosenbrock
  - Rastrigin
  - Offset Rastrigin
  - NK-Landscapes
  - DTrap

<table>
<thead>
<tr>
<th>Problem</th>
<th>Comparison</th>
<th>SCX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rosenbrock</td>
<td>-86.94 (54.54)</td>
<td>-26.47 (23.33)</td>
</tr>
<tr>
<td>Rastrigin</td>
<td>-59.2 (6.998)</td>
<td>-0.0088 (0.021)</td>
</tr>
<tr>
<td>Offset Rastrigin</td>
<td>-0.1175 (0.116)</td>
<td>-0.03 (0.028)</td>
</tr>
<tr>
<td>NK</td>
<td>0.771 (0.011)</td>
<td>0.8016 (0.013)</td>
</tr>
<tr>
<td>DTrap</td>
<td>0.9782 (0.005)</td>
<td>0.9925 (0.021)</td>
</tr>
</tbody>
</table>
Adaptations: Rastrigin
Adaptations: DTrap
SCX Overhead

• Requires No Additional Evaluation

• Adds No Significant Increase in Run Time
  – All linear operations

• Adds Initial Crossover Length Parameter
  – Testing showed results fairly insensitive to this parameter
  – Even worst settings tested achieved better results than comparison operators
Conclusions

• Remove Need to Select Crossover Algorithm

• Better Fitness Without Significant Overhead

• Benefits From Dynamically Changing Operator

• Promising Approach for Evolving Crossover Operators for Additional Representations (e.g., Permutations)
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.
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.
Case Study 3: The Automated Design of Mutation Operators
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.

• *Let's invent some more!!!*

• 😞 NO, let's build a general method (for problem class)

---

What probability distribution of problem instances are these intended
Off-the-Shelf metaheuristic to Tailor-Make mutation operators for Problem Class

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

Base-level
Genetic Algorithm
Fitness value
Mutation operator

search space

novel mutation heuristics

One Point mutation
Uniform mutation

Two search spaces
Commonly used
Mutation operators
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** \( R_i \leftarrow -1 * R_i \)
- **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>Ivt, −3</td>
<td>Ivt, −3</td>
</tr>
<tr>
<td>13</td>
<td>Nop</td>
<td>Nop</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 $t = -0.7 + 0.5 \, N(0, 1)$ in the range $[-1, +1]$.
3. We linearly interpolate the value $t$ from the range $[-1, +1]$ into an integer in the range $[0, 2^{\text{num-bits}} - 1]$, 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 Means and stand dev</th>
<th>Uniform Mutation</th>
<th>One-point mutation</th>
<th>generated-mutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1 mean</td>
<td>55.31</td>
<td>56.08</td>
<td>56.47</td>
</tr>
<tr>
<td>p1 std-dev</td>
<td>0.33</td>
<td>0.29</td>
<td>0.33</td>
</tr>
<tr>
<td>p2 mean</td>
<td>3064</td>
<td>3141</td>
<td>3168</td>
</tr>
<tr>
<td>p2 std-dev</td>
<td>33</td>
<td>35</td>
<td>33</td>
</tr>
<tr>
<td>p3 mean</td>
<td>2229</td>
<td>2294</td>
<td>2314</td>
</tr>
<tr>
<td>p3 std-dev</td>
<td>31</td>
<td>28</td>
<td>27</td>
</tr>
<tr>
<td>p4 mean</td>
<td>3065</td>
<td>3130</td>
<td>3193</td>
</tr>
<tr>
<td>p4 std-dev</td>
<td>36</td>
<td>24</td>
<td>28</td>
</tr>
<tr>
<td>p5 mean</td>
<td>0.839</td>
<td>0.846</td>
<td>0.861</td>
</tr>
<tr>
<td>p5 std-dev</td>
<td>0.012</td>
<td>0.01</td>
<td>0.012</td>
</tr>
<tr>
<td>p6 mean</td>
<td>0.643</td>
<td>0.643</td>
<td>0.663</td>
</tr>
<tr>
<td>p6 std-dev</td>
<td>0.004</td>
<td>0.004</td>
<td>0.003</td>
</tr>
<tr>
<td>p7 mean</td>
<td>0.752</td>
<td>0.7529</td>
<td>0.7684</td>
</tr>
<tr>
<td>p7 std-dev</td>
<td>0.0028</td>
<td>0.004</td>
<td>0.0031</td>
</tr>
<tr>
<td>class</td>
<td>32 bit</td>
<td>32 bit</td>
<td>64 bit</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>p1</td>
<td>1.98E-08</td>
<td>0.0005683</td>
<td>1.64E-19</td>
</tr>
<tr>
<td>p2</td>
<td>1.21E-18</td>
<td>1.08E-12</td>
<td>1.63E-17</td>
</tr>
<tr>
<td>p3</td>
<td>1.57E-17</td>
<td>1.65E-14</td>
<td>3.49E-16</td>
</tr>
<tr>
<td>p4</td>
<td>4.74E-23</td>
<td>1.22E-16</td>
<td>2.35E-21</td>
</tr>
<tr>
<td>p5</td>
<td>9.62E-17</td>
<td>1.67E-15</td>
<td>4.80E-09</td>
</tr>
<tr>
<td>p6</td>
<td>2.54E-27</td>
<td>4.14E-24</td>
<td>3.31E-24</td>
</tr>
<tr>
<td>p7</td>
<td>1.34E-24</td>
<td>3.00E-18</td>
<td>1.45E-28</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.
Case Study 4: The Automated Design of Black Box Search Algorithms [21, 23, 25]
Approach

• Hyper-Heuristic employing Genetic Programing

• Post-ordered parse tree

• Evolve the iterated function
Our Solution

Initialization

Check for Termination

Terminate

Iterated Function
Our Solution

• Hyper-Heuristic employing Genetic Programming

• Post-ordered parse tree

• Evolve the iterated function

• High-level primitives
• Iterated function

• Sets of solutions

• Function returns a set of solutions accessible to the next iteration
Primitive Types

• Variation Primitives
• Selection Primitives
• Set Primitives
• Evaluation Primitive
• Terminal Primitives
Variation Primitives

- **Bit-flip Mutation**
  - *rate*

- **Uniform Recombination**
  - *count*

- **Diagonal Recombination**
  - *n*
Selection Primitives

• Truncation Selection
  – count

• K-Tournament Selection
  – k
  – count

• Random Sub-set Selection
  – count
Set-Operation Primitives

- Make Set
  - name

- Persistent Sets
  - name

- Union
Evaluation Primitive

- Evaluates the nodes passed in

- Allows multiple operations and accurate selections within an iteration
  - Allows for deception
Terminal Primitives

• Random Individuals
  – *count*

• `Last` Set

• Persistent Sets
  – *name*
Meta-Genetic Program

1. Create Valid Population
2. Generate Children
3. Evaluate Children
4. Select Survivors
5. Check Termination

BBSA Evaluation

- Create Valid Population
- Generate Children
- Evaluate Children
- Select Survivors
- Generate Children

Termination Conditions

- Evaluations
- Iterations
- Operations
- Convergence
Proof of Concept Testing

- Deceptive Trap Problem

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Fitness vs. # of 1s diagram
Proof of Concept Testing (cont.)

• Evolved Problem Configuration
  – Bit-length = 100
  – Trap Size = 5

• Verification Problem Configurations
  – Bit-length = 100, Trap Size = 5
  – Bit-length = 200, Trap Size = 5
  – Bit-length = 105, Trap Size = 7
  – Bit-length = 210, Trap Size = 7
Results

<table>
<thead>
<tr>
<th>BBSA</th>
<th>EA</th>
<th>Hill-Climber</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>2</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>3</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>6</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>7</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>12</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>13</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>14</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>15</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

60% Success Rate
Results:

Bit-Length = 100
Trap Size = 5
Results:
Bit-Length = 200
Trap Size = 5
Results:
Bit-Length = 105
Trap Size = 7
Results:

Bit-Length = 210
Trap Size = 7

![Graph showing fitness over evaluations for different algorithms: EA, Hill, BBSA1, BBSA2, and BBSA3. The graph indicates how each algorithm performs relative to each other over a range of evaluations.](image-url)
Insights

• Diagonal Recombination
BBSA2

union

evaluate

diagonal
n: 3

mutate
rate: 0.993140497282

trunc
count: 15

count: 8

count: 17

last

last
Insights

• Diagonal Recombination

• Generalization
Insights

• Diagonal Recombination

• Generalization

• Over-Specialization
Over-Specialization

Trained Problem Configuration

Alternate Problem Configuration
Robustness

• Measures of Robustness
  – Applicability
  – Falliblility

• Applicability
  – What area of the problem configuration space do I perform well on?

• Fallibility
  – If a given BBSA doesn’t perform well, how much worse will I perform?
Robustness

![Graph showing the relationship between BBSA performance and problem configurations, highlighting the concept of applicability and fallibility.](image-url)
Multi-Sampling

• Train on multiple problem configurations

• Results in more robust BBSAs

• Provides the benefit of selecting the region of interest on the problem configuration landscape
Multi-Sample Testing

- Deceptive Trap Problem

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td># of 1s</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td># of 1s</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Fitness vs. # of 1s graph
Multi-Sample Testing (cont.)

• Multi-Sampling Evolution
  – Levels 1-5

• Training Problem Configurations
  1. Bit-length = 100, Trap Size = 5
  2. Bit-length = 200, Trap Size = 5
  3. Bit-length = 105, Trap Size = 7
  4. Bit-length = 210, Trap Size = 7
  5. Bit-length = 300, Trap Size = 5
Initial Test Problem Configurations

1. Bit-length = 100, Trap Size = 5
2. Bit-length = 200, Trap Size = 5
3. Bit-length = 105, Trap Size = 7
4. Bit-length = 210, Trap Size = 7
5. Bit-length = 300, Trap Size = 5
6. Bit-length = 99, Trap Size = 9
7. Bit-length = 198, Trap Size = 9
8. Bit-length = 150, Trap Size = 5
9. Bit-length = 250, Trap Size = 5
10. Bit-length = 147, Trap Size = 7
11. Bit-length = 252, Trap Size = 7
### Initial Results

<table>
<thead>
<tr>
<th>Level</th>
<th>Run</th>
<th>Train Fit.</th>
<th>Test Fit.</th>
<th>Fallibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1.0</td>
<td>0.976</td>
<td>0.094</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1.0</td>
<td>0.999</td>
<td>8.33 E-3</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0.944</td>
<td>0.883</td>
<td>0.082</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>0.976</td>
<td>0.894</td>
<td>0.224</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0.997</td>
<td>0.996</td>
<td>0.023</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0.992</td>
<td>0.959</td>
<td>0.130</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>0.966</td>
<td>0.970</td>
<td>0.054</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0.979</td>
<td>0.947</td>
<td>0.120</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0.965</td>
<td>0.966</td>
<td>0.050</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>0.984</td>
<td>0.980</td>
<td>0.065</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0.899</td>
<td>0.886</td>
<td>0.059</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0.926</td>
<td>0.898</td>
<td>0.073</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0.976</td>
<td>0.999</td>
<td>5.00 E-3</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>0.973</td>
<td>0.969</td>
<td>0.093</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>0.982</td>
<td>0.975</td>
<td>0.059</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0.993</td>
<td>0.999</td>
<td>5.00 E-3</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0.973</td>
<td>0.977</td>
<td>0.050</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>0.893</td>
<td>0.879</td>
<td>0.035</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>0.850</td>
<td>0.850</td>
<td>0.045</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>0.953</td>
<td>0.986</td>
<td>0.029</td>
</tr>
</tbody>
</table>
Problem Configuration Landscape Analysis

• Run evolved BBSAs on wider set of problem configurations

• Bit-length: ~75-~500

• Trap Size: 4-20
Results: Multi-Sampling Level 1
Results: Multi-Sampling Level 2
Results: Multi-Sampling Level 3
Results: Multi-Sampling Level 4
Results: Multi-Sampling Level 5
Results: EA Comparison
Discussion

• Robustness

  – Fallibility
Robustness: Fallibility

Multi-Sample Level 5

Standard EA
Robustness: Fallibility

Multi-Sample Level 1

Standard EA
Discussion

• Robustness

• Fallibility

• Applicability
Robustness: Applicability

Multi-Sample Level 1

Multi-Sample Level 5
Robustness: Applicability

<table>
<thead>
<tr>
<th>Level</th>
<th>Run</th>
<th>Train Fit.</th>
<th>Test Fit.</th>
<th>Fallibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1</td>
<td>0.973</td>
<td>0.977</td>
<td>0.050</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>0.893</td>
<td>0.879</td>
<td>0.035</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>0.850</td>
<td>0.850</td>
<td>0.045</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>0.955</td>
<td>0.986</td>
<td>0.029</td>
</tr>
</tbody>
</table>
Drawbacks

• Increased computational time
  – More runs per evaluation (increased wall time)
  – More problem configurations to optimize for (increased evaluations)
Summary of Multi-Sample Improvements

• Improved Hyper-Heuristic to evolve more robust BBSAs

• Evolved custom BBSA which outperformed standard EA and were robust to changes in problem configuration
Case Study 5: The Automated Design of Mutation Operators for Evolutionary Programming
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)
Optimization & Benchmark Functions

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

We use them as problem classes.

<table>
<thead>
<tr>
<th>Test function</th>
<th>( n )</th>
<th>( S )</th>
<th>( f_{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}</td>
<td>x_i + 0.5</td>
<td>)</td>
<td>30</td>
</tr>
<tr>
<td>( f_7(x) = \sum_{i=1}^{n} ix_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>$S$</th>
<th>$b$</th>
<th>$f_{\text{min}}$</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 \mid x_i \mid, 1 \leq i \leq n}$</td>
<td>$[-100, 100]^n$</td>
<td>N/A</td>
<td>0</td>
</tr>
<tr>
<td>$f_5(x) = \sum_{i=1}^{n} [a(x_{i+1} - x_i^2)^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} \left(\frac{ax_i + 0.5}{2}\right)^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)

Evolutionary Programming
\((R^n \rightarrow R) \rightarrow R^n\)

**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)

Evolutionary Programming Designer
\([((R^n \rightarrow R)] \rightarrow ((R^n \rightarrow R) \rightarrow R^n)]\)

**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.

8 May, 2015  John R. Woodward, Daniel R. Tauritz
Genetic Programming to Generate Probability Distributions

1. **GP Function Set** {+, -, *, %}
2. **GP Terminal Set** {N(0, 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</th>
<th>CEP</th>
<th>GP-distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean Best</td>
<td>Std Dev</td>
<td>Mean Best</td>
</tr>
<tr>
<td>$f_1$</td>
<td>$1.24 \times 10^{-3}$</td>
<td>$2.69 \times 10^{-4}$</td>
<td>$1.45 \times 10^{-4}$</td>
</tr>
<tr>
<td>$f_2$</td>
<td>$1.53 \times 10^{-1}$</td>
<td>$2.72 \times 10^{-2}$</td>
<td>$4.30 \times 10^{-2}$</td>
</tr>
<tr>
<td>$f_3$</td>
<td>$2.74 \times 10^{-2}$</td>
<td>$2.43 \times 10^{-2}$</td>
<td>$5.15 \times 10^{-2}$</td>
</tr>
<tr>
<td>$f_4$</td>
<td>$1.79$</td>
<td>$1.84$</td>
<td>$1.75 \times 10$</td>
</tr>
<tr>
<td>$f_5$</td>
<td>$2.52 \times 10^{-3}$</td>
<td>$4.96 \times 10^{-4}$</td>
<td>$2.66 \times 10^{-4}$</td>
</tr>
<tr>
<td>$f_6$</td>
<td>$3.86 \times 10^{-2}$</td>
<td>$3.12 \times 10^{-2}$</td>
<td>$4.40 \times 10$</td>
</tr>
<tr>
<td>$f_7$</td>
<td>$6.49 \times 10^{-2}$</td>
<td>$1.04 \times 10^{-2}$</td>
<td>$6.64 \times 10^{-2}$</td>
</tr>
<tr>
<td>$f_8$</td>
<td>$-11342.0$</td>
<td>$3.26 \times 10^{2}$</td>
<td>$-7894.6$</td>
</tr>
<tr>
<td>$f_9$</td>
<td>$6.24 \times 10^{-2}$</td>
<td>$1.30 \times 10^{-2}$</td>
<td>$1.09 \times 10^{2}$</td>
</tr>
<tr>
<td>$f_{10}$</td>
<td>$1.67$</td>
<td>$4.26 \times 10^{-1}$</td>
<td>$1.45$</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>20000</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>
## Performance on Other Problem Classes

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>ADR1</th>
<th>ADR2</th>
<th>ADR3</th>
<th>ADR4</th>
<th>ADR5</th>
<th>ADR6</th>
<th>ADR7</th>
<th>ADR8</th>
<th>ADR9</th>
<th>ADR10</th>
<th>ADR11</th>
<th>ADR12</th>
<th>ADR13</th>
<th>ADR14</th>
<th>ADR15</th>
<th>ADR16</th>
<th>ADR17</th>
<th>ADR18</th>
<th>ADR19</th>
<th>ADR20</th>
<th>ADR21</th>
<th>ADR22</th>
<th>ADR23</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.329435723</td>
<td>2.949197635</td>
<td>1.328093567</td>
<td>1.328093567</td>
<td>1.328093567</td>
<td>1.328093567</td>
<td>1.328093567</td>
<td>1.328093567</td>
<td>1.328093567</td>
<td>1.328093567</td>
<td>1.328093567</td>
<td>1.328093567</td>
<td>1.328093567</td>
<td>1.328093567</td>
<td>1.328093567</td>
<td>1.328093567</td>
<td>1.328093567</td>
<td>1.328093567</td>
<td>1.328093567</td>
<td>1.328093567</td>
<td>1.328093567</td>
<td>1.328093567</td>
<td></td>
</tr>
<tr>
<td>0.323722533</td>
<td>0.219590374</td>
<td>0.306543210</td>
<td>0.221139865</td>
<td>0.221139865</td>
<td>0.221139865</td>
<td>0.221139865</td>
<td>0.221139865</td>
<td>0.221139865</td>
<td>0.221139865</td>
<td>0.221139865</td>
<td>0.221139865</td>
<td>0.221139865</td>
<td>0.221139865</td>
<td>0.221139865</td>
<td>0.221139865</td>
<td>0.221139865</td>
<td>0.221139865</td>
<td>0.221139865</td>
<td>0.221139865</td>
<td>0.221139865</td>
<td>0.221139865</td>
<td></td>
</tr>
<tr>
<td>1.027255906</td>
<td>0.604118845</td>
<td>0.247589830</td>
<td>0.222059292</td>
<td>0.322163677</td>
<td>0.141322551</td>
<td>10.8830164</td>
<td>0.225013984</td>
<td>0.225013984</td>
<td>0.225013984</td>
<td>0.225013984</td>
<td>0.225013984</td>
<td>0.225013984</td>
<td>0.225013984</td>
<td>0.225013984</td>
<td>0.225013984</td>
<td>0.225013984</td>
<td>0.225013984</td>
<td>0.225013984</td>
<td>0.225013984</td>
<td>0.225013984</td>
<td>0.225013984</td>
<td></td>
</tr>
</tbody>
</table>

Performance on Other Problem Classes

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.
Case Study 6: The Automated Design of On-Line Bin Packing Algorithms
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

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, Daniel R. Tauritz
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?).
Hyper-heuristics Visualization Demo
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.
3. Let metaheuristics (e.g. Genetic Programming) search for 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
4. Data Mining – Gisele L. Pappa, Alex A. Freitas
5. Decision Tree - Gisele L. Pappa et al
7. Selection Heuristics – Woodward & Swan, Daniel Tauritz et al
8. Bin Packing 1,2,3 dimension (on and off line) Edmund Burke et. al. & Riccardo Poli et al
9. Bug Location – Shin Yoo
10. 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.

Human cost (INFLATION)  machine cost MOORE’S LAW

conventional approach  new approach

• 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.
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</th>
<th>BIN PACKING</th>
<th>MUTATION</th>
<th>Crossover</th>
<th>BBSA</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scalable performance</strong></td>
<td>Not yet tested</td>
<td>Not yet tested</td>
<td>Yes - why</td>
<td>No - why</td>
<td>Not yet tested</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Generation zero human comp.</strong></td>
<td>Rank, fitness proportional</td>
<td>No – needed to seed</td>
<td>Best fit</td>
<td>Gaussian and Cauchy</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>Problem classes tested</strong></td>
<td>Shifted function</td>
<td>Parameterized function</td>
<td>Item size</td>
<td>Parameterized function</td>
<td>Rosenbrock, NK-Landscapes, Rastrigin, etc.</td>
<td>DTrap, NK-landscapes</td>
</tr>
<tr>
<td><strong>Results Human Competitive</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Algorithm iterate over</strong></td>
<td>Population</td>
<td>Bit-string</td>
<td>Bins</td>
<td>Vector</td>
<td>Pair of parents</td>
<td>Population</td>
</tr>
<tr>
<td><strong>Search Method</strong></td>
<td>Random Search</td>
<td>Iterative Hill-Climber</td>
<td>Genetic Programming</td>
<td>Genetic Programming</td>
<td>Linear Genetic Programming</td>
<td>Tree-based GP</td>
</tr>
<tr>
<td><strong>Type Signatures</strong></td>
<td>R^2-&gt;R</td>
<td>B^n-&gt;B^n</td>
<td>R^3-&gt;R</td>
<td>()-&gt;R</td>
<td>R^n -&gt; R^m</td>
<td>Population -&gt; Population</td>
</tr>
<tr>
<td><strong>Reference</strong></td>
<td>[16]</td>
<td>[15]</td>
<td>[6,9,10,11]</td>
<td>[18]</td>
<td>[20]</td>
<td>[21,23,25]</td>
</tr>
</tbody>
</table>
SUMMARY

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,**
7. improve existing algorithms and adapt them to the new problem instances.
8. Humans are working at a **higher level of abstraction and more creative**. Creating search spaces for GP to sample.
9. **Algorithms are reusable, “solutions” aren’t**. (e.g. tsp algorithm vs route)
10. Opens up new problem domains. E.g. bin-packing.
Related hyper-heuristics events

• Evolutionary Computation for the Automated Design of Algorithms (ECADA) workshop @GECCO 2015

• Combinatorial Black Box Optimization Competition (CBBOC) @GECCO 2015
• Thank you for listening !!!
• We are glad to take any
  – comments (+,-)
  – suggestions/criticisms
Please email us any missing references!
John Woodward (http://www.cs.stir.ac.uk/~jrw/)
Daniel Tauritz (http://web.mst.edu/~tauritzd/)


References 2


References 3


References 4


