Hyper-heuristics

Daniel R. Tauritz (dtauritz@acm.org)

Natural Computation Laboratory, Missouri University of Science and Technology (http://web.mst.edu/~tauritzd/nc-lab/)

John Woodward (John.Woodward@cs.stir.ac.uk)

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

ACM 978-1-4503-4323-7/16/07. http://dx.doi.org/10.1145/2908961.2926978

Instructors

Daniel R. Tauritz is an Associate Professor in the Department of Computer Science at the Missouri University of Science and Technology (S&T), a contract scientist for Sandia National Laboratories, a former Guest Scientist at Los Alamos National Laboratory (LANL), the founding director of S&T's Natural Computation Laboratory, and founding academic director of the LANL/S&T Cyber Security Sciences Institute. He received his Ph.D. in 2002 from Leiden University. His research interests include the design of hyper-heuristics and self-configuring evolutionary algorithms and the application of computational intelligence techniques in cyber security, critical infrastructure protection, and program understanding.

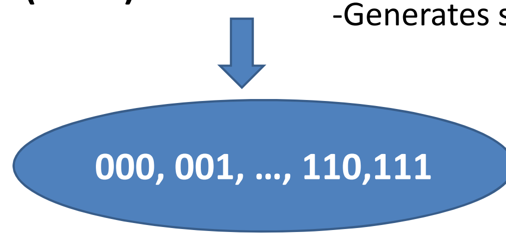
John R. Woodward is a Lecturer at the University of Stirling, within the CHORDS group and is employed on the DAASE project, and for the previous four years was a lecturer with the University of Nottingham. He holds a BSc in Theoretical Physics, an MSc in Cognitive Science and a PhD in Computer Science, all from the University of Birmingham. His research interests include Automated Software Engineering, particularly Search Based Software Engineering, Artificial Intelligence/Machine Learning and in particular Genetic Programming. He has worked in industrial, military, educational and academic settings, and been employed by EDS, CERN and RAF and three UK Universities.

John's perspective of hyper-heuristics

Metaheuristic

A metaheuristic
- Sample solutions
- Generates solutions

(Meta)heuristic

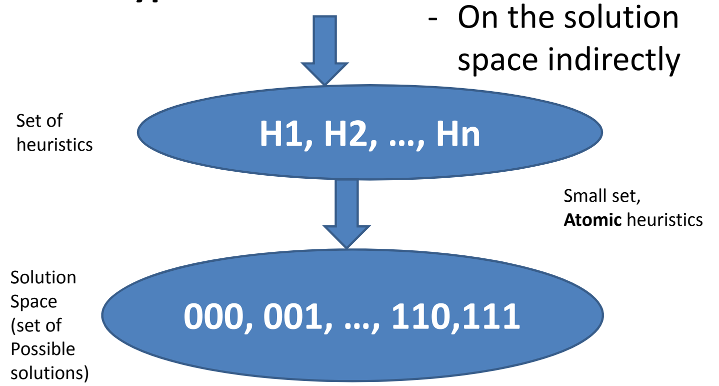


Solution Space (set of Possible solutions)

Hyper-heuristic

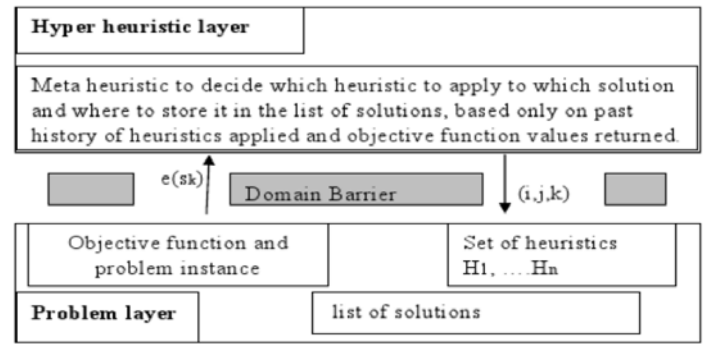
Hyper-heuristic operated on
- Other heuristics
- On the solution space indirectly

Hyper-heuristic



Domain Barrier

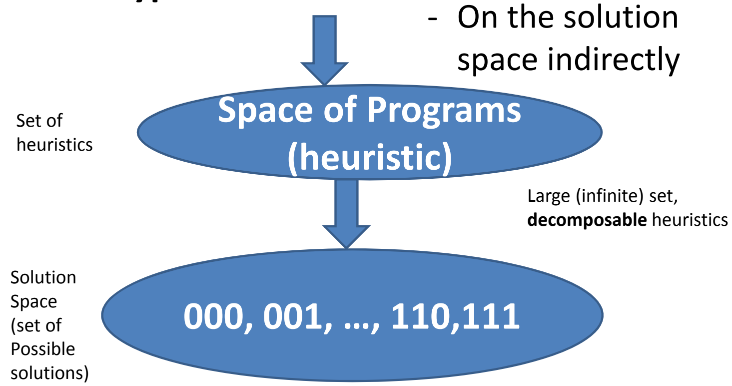
$$HH : (H \times \mathbb{N}^2 \times \mathbb{R})^* \rightarrow H \times \mathbb{N}^2$$



Hyper-heuristic

Hyper-heuristic operated on
- Other heuristics
- On the solution space indirectly

Hyper-heuristic



Performance table

TABLE I
A CLASSIFICATION TABLE SHOWING ALGORITHMS IN COLUMNS AND PROBLEMS ROWS, AND THE PERFORMANCE OF THAT ALGORITHM ON THAT PROBLEM IN THE INTERSECTING CELL.

	algorithm-1	algorithm-2	…	algorithm-m
problem-1	1.0	2.3	…	3.6
problem-2	1.6	6.5	…	7.6
…	…	…	…	…
problem-n	3.5	8.6	…	6.5

	$f_{<y_1,y_2>}$	$f_{<y_1,y_1>}$	$f_{<y_2,y_2>}$	$f_{<y_2,y_1>}$
$< x_1, x_2 >$	$< y_1, y_2 >$	$< y_1, y_1 >$	$< y_2, y_2 >$	$< y_2, y_1 >$
$< x_2, x_1 >$	$< y_2, y_1 >$	$< y_1, y_1 >$	$< y_2, y_2 >$	$< y_1, y_2 >$

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

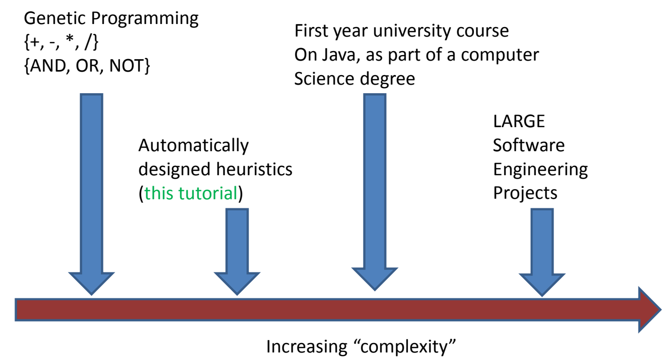
TSP algorithm

EXECUTABLE on MANY INSTANCES!!!

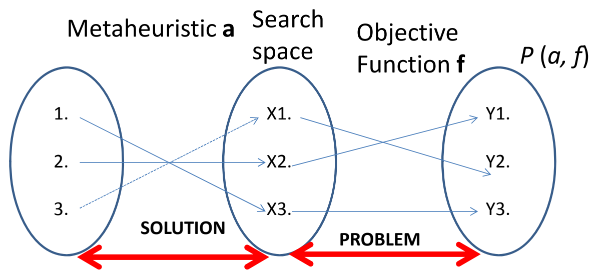
Give a man a fish and he will eat for a **day**.
Teach a man to fish and he will eat for a **lifetime.**

Scalable? General?
New domains for GP

Program-Complexity Spectrum

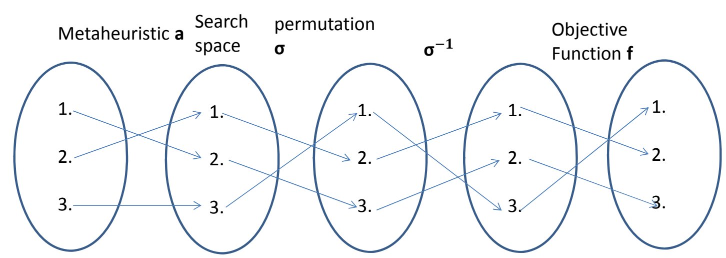


Theoretical Motivation 1



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 a (approximate) permutation. What are we learning?
4. **Performance measure** *P* (*a*, *f*) depend only on y1, y2, y3
5. Aim **find a solution with a near-optimal objective value using a Metaheuristic .** ANY QUESTIONS BEFORE NEXT SLIDE?

Theoretical Motivation 2



$P(a, f) = P(a\ \boldsymbol{\sigma}, \boldsymbol{\sigma}^{-1} f)$ $P(A, F) = P(A\boldsymbol{\sigma}, \boldsymbol{\sigma}^{-1} F)$ (i.e. permute bins)
P is a **performance measure**, (based only on output values).
$\boldsymbol{\sigma}, \boldsymbol{\sigma}^{-1}$ 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 $\boldsymbol{\sigma\sigma^{-1} f}$ (that is $\boldsymbol{f}$)
2. Metaheuristic **aσ** applied to function $\boldsymbol{\sigma^{-1} f}$ 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.

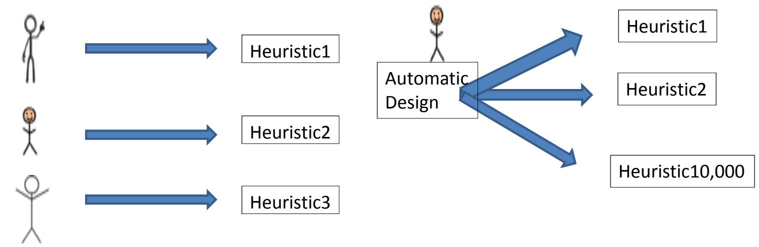
One Man – One/Many Algorithm

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]

Type of GP Matters: Experiment Description

- Implement five types of GP (tree GP, linear GP, canonical Cartesian GP, Stack GP, and Grammatical Evolution) in hyper-heuristics for evolving black-box search algorithms for solving 3-SAT
- Base hyper-heuristic fitness on the fitness of the best search algorithm generated at solving the 3-SAT problem
- Compare relative effectiveness of each GP type as a hyper-heuristic

GP Individual Description

- Search algorithms are represented as an iterative algorithm that passes one or more set of variable assignments to the next iteration
- Genetic program represents a single program iteration
- Algorithm runs starting with a random initial population of solutions for 30 seconds

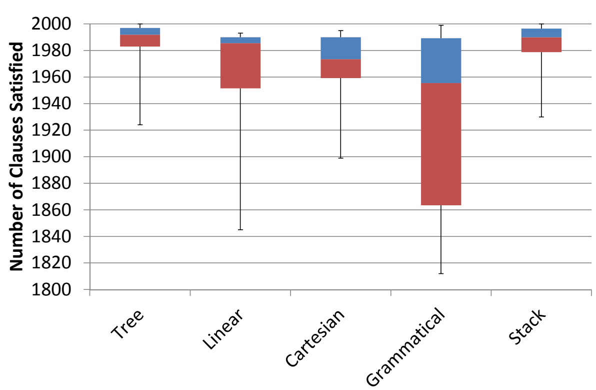
3-SAT Problem

- A subset of the Boolean Satisfiability Problem (SAT)
- The goal is to select values for Boolean variables such that a given Boolean equation evaluates as true (is satisfied)
- Boolean equations are in 3-conjunctive normal form
- Example:
  – (A ∨ B ∨ C) ∧ (¬A ∨ ¬C ∨ D) ∧ (¬B ∨ C ∨ ¬D)
  – Satisfied by ¬A, B, C, ¬D
- Fitness is the number of clauses satisfied by the best solution in the final population

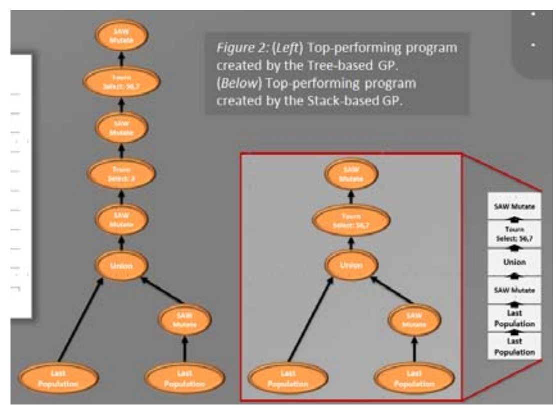
Genetic Programming Nodes Used

- Last population, Random population
- Tournament selection, Fitness proportional selection, Truncation selection, Random selection
- Bitwise mutation, Greedy flip, Quick greedy flip, Stepwise adaption of weights, Novelty
- Union

Results



Results [24]



24

Results

- Generated algorithms mostly performed comparably well on training and test problems
- Tree and stack GP perform similarly well on this problem, as do linear and Cartesian GP
- Tree and stack GP perform significantly better on this problem than linear and Cartesian GP, which perform significantly better than grammatical evolution

Conclusions

- The choice of GP type makes a significant difference in the performance of the hyper-heuristic
- The size of the search space appears to be a major factor in the performance of the hyper-heuristic

Case Study 1: The Automated Design of Crossover Operators [20]

Motivation

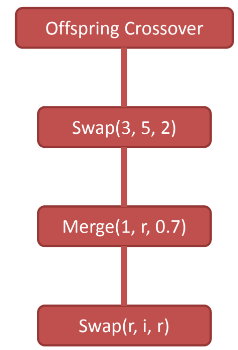
- Performance Sensitive to Crossover Selection
- Identifying & Configuring Best Traditional Crossover is Time Consuming
- Existing Operators May Be Suboptimal
- Optimal Operator May Change During Evolution

Some Possible Solutions

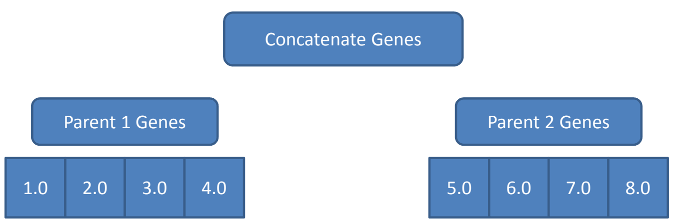
- Meta-EA
  - Exceptionally time consuming
- Self-Adaptive Algorithm Selection
  - Limited by algorithms it can choose from

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



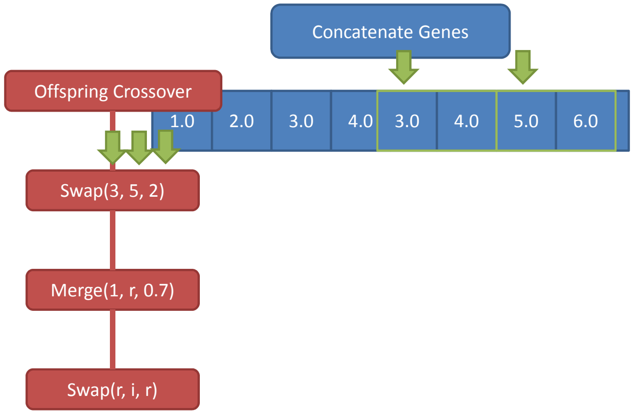
Applying an SCX



The Swap Primitive

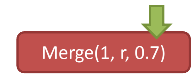
- Each Primitive has a type
  - Swap represents crossovers that move genetic material
- First Two Parameters
  - Start 1 Position
  - Start 2 Position
- Third Parameter Primitive Dependent
  - Swaps use "Width"

Applying an SCX



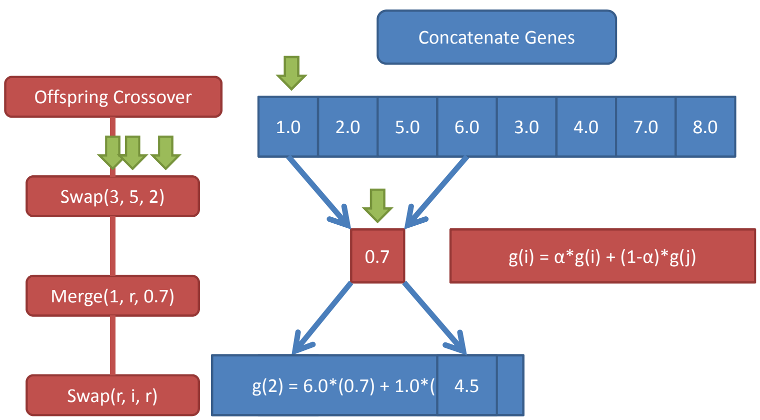
The Merge Primitive

- Third Parameter Primitive Dependent
  - Merges use "Weight"



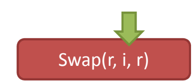
- Random Construct
  - All past primitive parameters used the Number construct
  - "r" marks a primitive using the Random Construct
  - Allows primitives to act stochastically

Applying an SCX



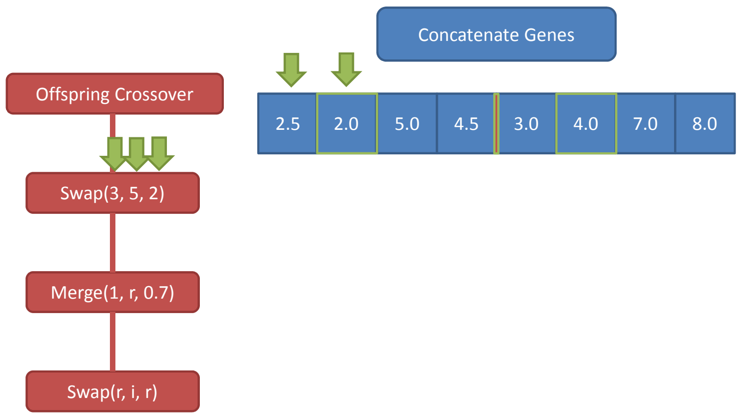
The Inline Construct

- Only Usable by First Two Parameters

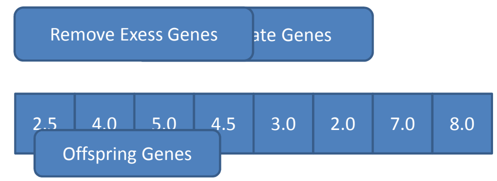


- Denoted as "i"
- Forces Primitive to Act on the Same Loci in Both Parents

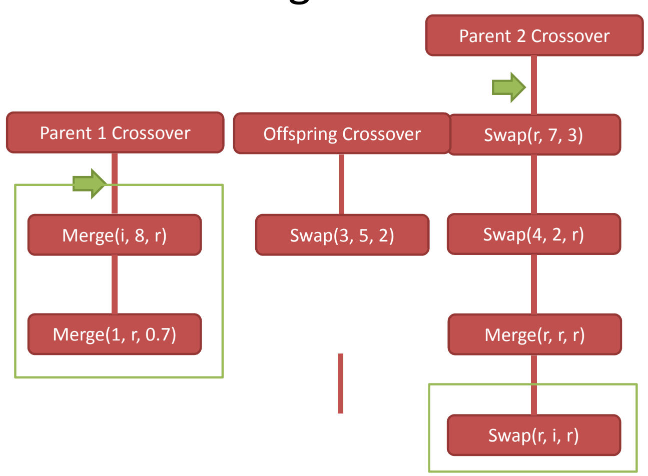
Applying an SCX



Applying an SCX



Evolving Crossovers

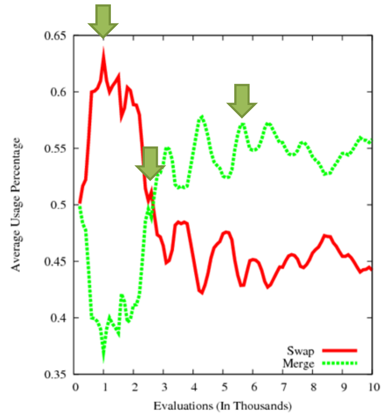


Empirical Quality Assessment

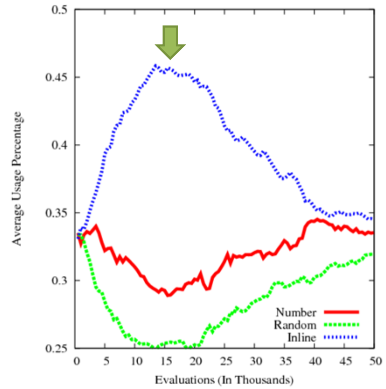
- Compared Against
  - Arithmetic Crossover
  - N-Point Crossover
  - Uniform Crossover
- On Problems
  - Rosenbrock
  - Rastrigin
  - Offset Rastrigin
  - NK-Landscapes
  - DTrap

Problem	Comparison	SCX
Rosenbrock	-86.94 (54.54)	-26.47 (23.33)
Rastrigin	-59.2 (6.998)	-0.0088 (0.021)
Offset Rastrigin	-0.1175 (0.116)	-0.03 (0.028)
NK	0.771 (0.011)	0.8016 (0.013)
DTrap	0.9782 (0.005)	0.9925 (0.021)

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

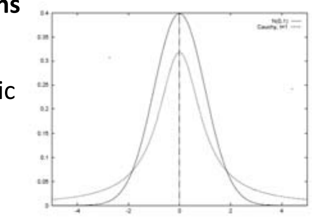
- Remove Need to Select Crossover Algorithm
- Better Fitness Without Significant Overhead
- Benefits From Dynamically Changing Operator
- Promising Approach for Evolving Crossover Operators for Additional Representations (e.g., Permutations)

Case Study 2: The Automated Design of Mutation Operators for Evolutionary Programming

Designing Mutation Operators for Evolutionary Programming [18]

1. **Evolutionary programing** 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** (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 μ individuals, and set $k = 1$. Each individual is taken as a pair of real-valued vectors, (x_i, η_i), $\forall i \in \{1, \cdots, \mu\}$.
2. Evaluate the fitness score for each individual (x_i, η_i), $\forall i \in \{1, \cdots, \mu\}$, of the population based on the objective function, $f(x_i)$.
3. Each parent (x_i, η_i), $i = 1, \cdots, \mu$, creates a single offspring (x_i', η_i') by: for $j = 1, \cdots, n$,
$$x_i'(j) = x_i(j) + \eta_i(j)N(0,1), \quad (1)$$
$$\eta_i'(j) = \eta_i(j)\exp(\tau' N(0,1) + \tau N_j(0,1)) \quad (2)$$
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', η_i and η_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 τ and τ' have commonly set to $\left(\sqrt{2\sqrt{n}}\right)^{-1}$ and $\left(\sqrt{2n}\right)^{-1}$ [9, 8].
4. Calculate the fitness of each offspring (x_i', η_i'), $\forall i \in \{1, \cdots, \mu\}$.
5. Conduct pairwise comparison over the union of parents (x_i, η_i) and offspring (x_i', η_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 μ individuals out of (x_i, η_i) and (x_i', η_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** $\forall x \in S : f(x_{min}) \leq f(x)$

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_{min} the minimum value of the function, and $S \subseteq R^n$.

Test function	n	S	f_{min}
$f_1(x) = \sum_{i=1}^n x_i^2$	30	$[-100, 100]^n$	0
$f_2(x) = \sum_{i=1}^n \lvert x_i \rvert + \prod_{i=1}^n \lvert x_i \rvert$	30	$[-10, 10]^n$	0
$f_3(x) = \sum_{i=1}^n (\sum_{j=1}^i x_j)^2$	30	$[-100, 100]^n$	0
$f_4(x) = \max_i\{\lvert x_i \rvert, 1 \leq i \leq n\}$	30	$[-100, 100]^n$	0
$f_5(x) = \sum_{i=1}^{n-1} [100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2]$	30	$[-30, 30]^n$	0
$f_6(x) = \sum_{i=1}^n \lfloor x_i + 0.5 \rfloor$	30	$[-100, 100]^n$	0
$f_7(x) = \sum_{i=1}^n ix_i^4 + random[0, 1)$	30	$[-1.28, 1.28]^n$	0
$f_8(x) = \sum_{i=1}^n -x_i \sin(\sqrt{\lvert x_i \rvert})$	30	$[-500, 500]^n$	-12569.5
$f_9(x) = \sum_{i=1}^n [x_i^2 - 10\cos(2\pi x_i) + 10)]$	30	$[-5.12, 5.12]^n$	0
$f_{10}(x) = -20\exp\left(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^n x_i^2}\right) - \exp\left(\frac{1}{n}\sum_{i=1}^n \cos 2\pi x_i\right) +20 + e$	30	$[-32, 32]^n$	0

Function Class 1

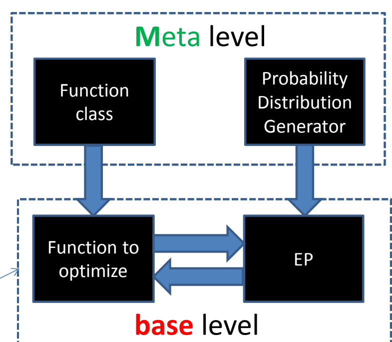
1. Machine learning needs to generalize.
2. We generalize to function classes.
3. y = x^2 (**a function**)
4. y = ax^2(parameterised function)
5. y = ax^2, a ~[1,2] (**function class**)
6. We do this for all benchmark functions.
7. **The mutation operators is evolved to fit the probability distribution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

Function Classes 2

Function Classes	S	b	f_{min}
$f_1(x) = a\sum_{i=1}^{n} x_i^2$	$[-100, 100]^n$	N/A	0
$f_2(x) = a\sum_{i=1}^{n} \vert x_i \vert + b\prod_{i=1}^{n} \vert x_i \vert$	$[-10, 10]^n$	$b \in [0, 10^{-5}]$	0
$f_3(x) = \sum_{i=1}^{n}(a\sum_{j=1}^{i} x_j)^2$	$[-100, 100]^n$	N/A	0
$f_4(x) = \max_i\{a \vert x_i \vert, 1 \leq i \leq n\}$	$[-100, 100]^n$	N/A	0
$f_5(x) = \sum_{i=1}^{n}[a(x_{i+1} - x_i^2)^2 + (x_i - 1)^2]$	$[-30, 30]^n$	N/A	0
$f_6(x) = \sum_{i=1}^{n}(\lfloor ax_i + 0.5 \rfloor)^2$	$[-100, 100]^n$	N/A	0
$f_7(x) = a\sum_{i=1}^{n} ix_i^4 + random[0, 1)$	$[-1.28, 1.28]^n$	N/A	0
$f_8(x) = \sum_{i=1}^{n} -(x_i \sin(\sqrt{\vert x_i \vert}) + a)$	$[-500, 500]^n$	N/A	[-12629.5, -12599.5]
$f_9(x) = \sum_{i=1}^{n}[ax_i^2 + b(1 - cos(2\pi x_i))]$	$[-5.12, 5.12]^n$	$b \in [5,\ 10]$	0
$f_{10}(x) = -a\exp(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2})$ $- \exp(\frac{1}{n}\sum_{i=1}^{n} \cos 2\pi x_i) + a + e$	$[-32, 32]^n$	N/A	0

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	Evolutionary Programming Designer
$(R^n \rightarrow R) \rightarrow R^n$	$[(R^n \rightarrow R)] \rightarrow ((R^n \rightarrow R) \rightarrow R^n)$
Input is a function mapping real-valued vectors of length n to a real-value.	**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) real-valued vector	**Output** is a (near optimal) (mutation operator for) Evolutionary Programming
(i.e. the solution to the problem instance)	(i.e. the solution method to the problem class)

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

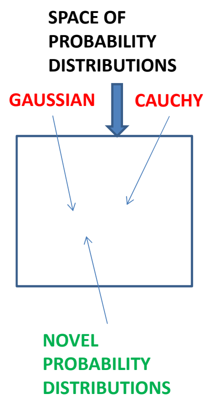
Genetic Programming to Generate Probability Distributions

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

Function Class	FEP Mean Best	FEP Std Dev	CEP Mean Best	CEP Std Dev	GP-distribution Mean Best	GP-distribution Std Dev
f_1	1.24×10^{-3}	2.69×10^{-4}	1.45×10^{-4}	9.95×10^{-5}	$\mathbf{6.37\times10^{-5}}$	5.56×10^{-5}
f_2	1.53×10^{-1}	2.72×10^{-2}	4.30×10^{-2}	9.08×10^{-3}	$\mathbf{8.14\times10^{-4}}$	8.50×10^{-4}
f_3	2.74×10^{-2}	2.43×10^{-2}	5.15×10^{-2}	9.52×10^{-2}	$\mathbf{6.14\times10^{-3}}$	8.78×10^{-3}
f_4	1.79	1.84	1.75×10	6.10	$\mathbf{2.16\times10^{-1}}$	6.54×10^{-1}
f_5	2.52×10^{-3}	4.96×10^{-4}	2.66×10^{-4}	4.65×10^{-5}	$\mathbf{8.39\times10^{-7}}$	1.43×10^{-7}
f_6	3.86×10^{-2}	3.12×10^{-2}	4.40×10	1.42×10^{2}	$\mathbf{9.20\times10^{-3}}$	1.34×10^{-2}
f_7	6.49×10^{-2}	1.04×10^{-2}	6.64×10^{-2}	1.21×10^{-2}	$\mathbf{5.25\times10^{-2}}$	8.46×10^{-3}
f_8	-11342.0	3.26×10^{2}	-7894.6	6.14×10^{2}	**-12611.6**	2.30×10
f_9	6.24×10^{-2}	1.30×10^{-2}	1.09×10^{2}	3.58×10	$\mathbf{1.74\times10^{-3}}$	4.25×10^{-4}
f_{10}	1.67	4.26×10^{-1}	1.45	2.77×10^{-1}	**1.38**	2.45×10^{-1}

T-tests

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

Function Class	Number of Generations	GP-distribution vs FEP *t-test*	GP-distribution vs CEP *t-test*
f_1	1500	2.78×10^{-47}	4.07×10^{-2}
f_2	2000	5.53×10^{-62}	1.59×10^{-54}
f_3	5000	8.03×10^{-8}	1.14×10^{-3}
f_4	5000	1.28×10^{-7}	3.73×10^{-36}
f_5	20000	2.80×10^{-58}	9.29×10^{-63}
f_6	1500	1.85×10^{-8}	3.11×10^{-2}
f_7	3000	3.27×10^{-9}	2.00×10^{-9}
f_8	9000	7.99×10^{-48}	5.82×10^{-75}
f_9	5000	6.37×10^{-55}	6.54×10^{-39}
f_{10}	1500	9.23×10^{-5}	1.93×10^{-1}

Performance on Other Problem Classes

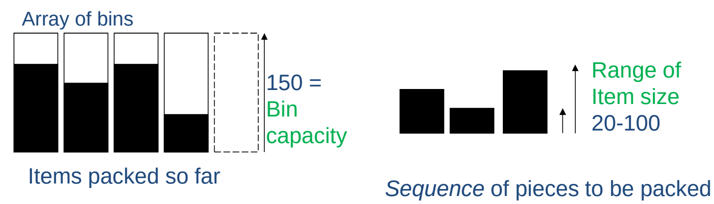
Table 8: This table compares the fitness values (averaged over 20 runs) of each of the 23 ADRs on each of the 23 function classes. Standard deviations are in parentheses.

[illegible]

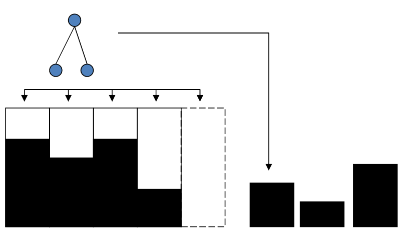
Case Study 3: The Automated Design of On-Line Bin Packing Algorithms

On-line Bin Packing Problem [9,11]

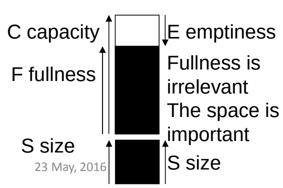
- A *sequence* of items packed into as few a bins as possible.
- Bin size is 150 units, items uniformly distributed between 20-100.
- Different to the off-line bin packing problem where the *set* of items.
- The "best fit" heuristic, places the current item in the space it fits best (leaving least slack).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



Genetic Programming applied to on-line bin packing

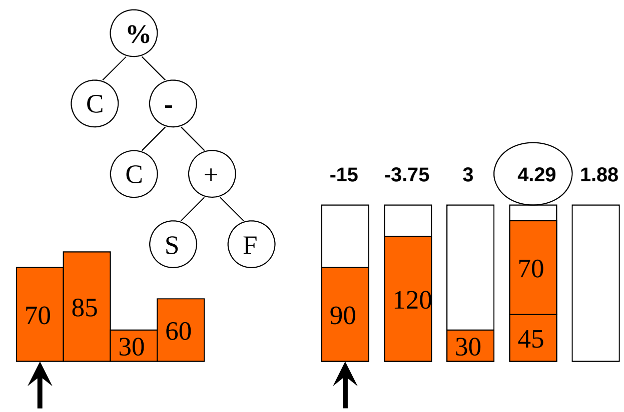


Not obvious how to link Genetic Programming to combinatorial problems.
The GP tree is applied to each bin with the current item and placed in the bin with The maximum score



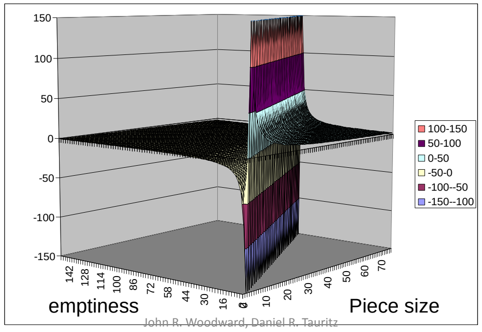
Terminals supplied to Genetic Programming
Initial representation {C, F, S}
Replaced with {E, S}, E=C-F

How the heuristics are applied (skip)

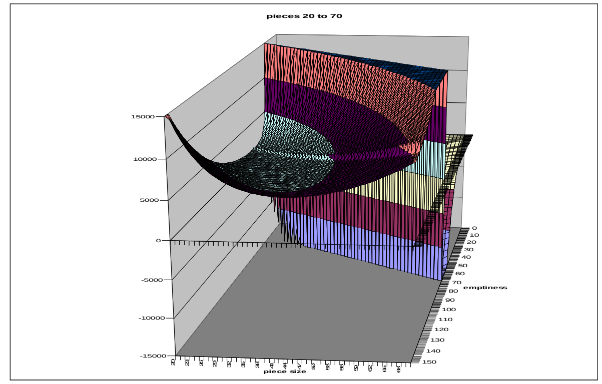


The Best Fit Heuristic

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.

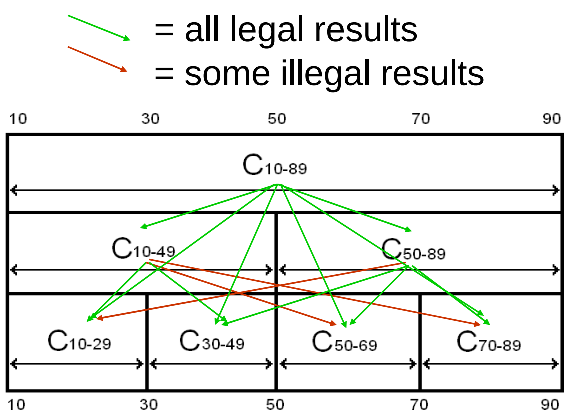


Our Best Heuristic



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

Robustness of Heuristics



Testing Heuristics on problems of much larger size than in training

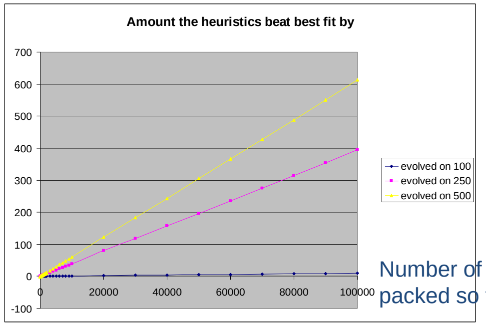
Table I	*H trained100*	*H trained 250*	*H trained 500*
100	0.427768358	0.298749035	0.140986023
1000	0.406790534	0.010006408	0.000350265
10000	0.454063071	2.58E-07	9.65E-12
100000	0.271828318	1.38E-25	2.78E-32

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

Amount evolved heuristics beat best fit by.

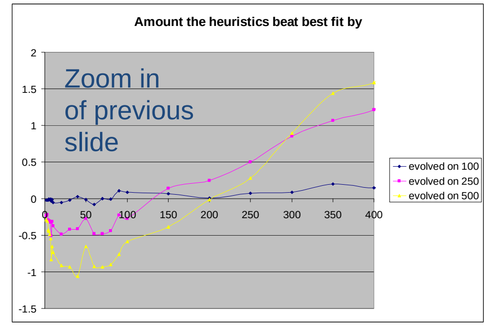


Number of pieces packed so far.

- 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

Amount evolved heuristics beat best fit by.



- 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?).

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").

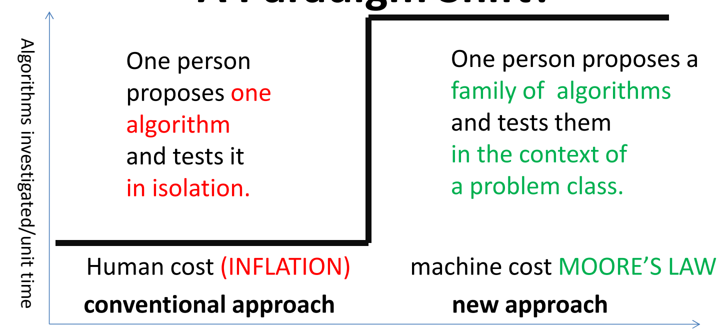
A Brief History (Example Applications) [5]

1. **Image Recognition** – Roberts Mark
2. **Travelling Salesman Problem** – Keller Robert
3. **Boolean Satisfiability** – Holger Hoos, Fukunaga, Bader-El-Den, Alex Bertels & Daniel Tauritz
4. **Data Mining** – Gisele L. Pappa, Alex A. Freitas
5. **Decision Tree** - Gisele L. Pappa et al
6. **Crossover Operators** – Oltean et al, Brian Goldman and Daniel Tauritz
7. **Selection Heuristics** – Woodward & Swan, Matthew Martin & Daniel Tauritz
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
11. **Black Box Search Algorithms** – Daniel Tauritz et al

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 produces 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?



- 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).

SUMMARY

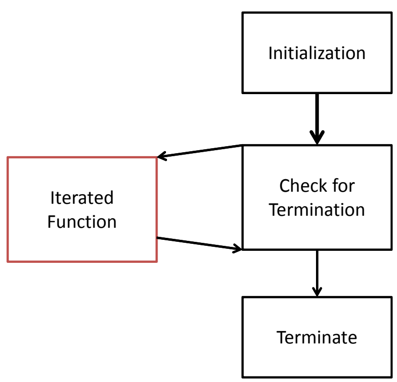
1. We can automatically design algorithms that **consistently outperform human designed algorithms (on various domains)**.
2. Humans should not provide variations– genetic programing can do that.
3. We are altering the heuristic to suit the set of problem instances presented to it, in the hope that it 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.

Case Study 4: The Automated Design of Black Box Search Algorithms [21, 23, 25]

Approach

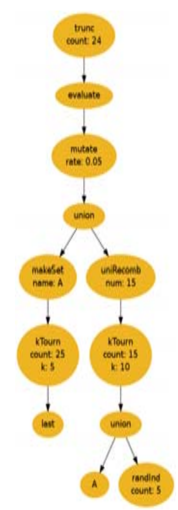
- Hyper-Heuristic employing Genetic Programing
- Post-ordered parse tree
- Evolve the iterated function

Our Solution

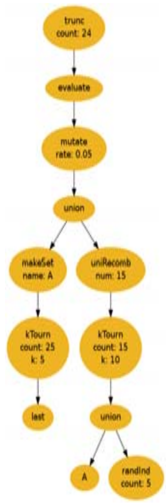


Our Solution

- Hyper-Heuristic employing Genetic Programing
- Post-ordered parse tree
- Evolve the iterated function
- High-level primitives



Parse Tree



- 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*

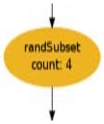
- Diagonal Recombination
  - *n*

Selection Primitives

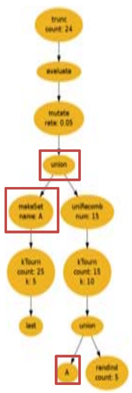
- Truncation Selection
  - *count*

- K-Tournament Selection
  - *k*
  - *count*

- Random Sub-set Selection
  - *count*



Set-Operation Primitives



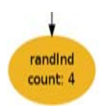
- Make Set
  - *name*
- Persistent Sets
  - *name*
- Union

Evaluation Primitive

- Evaluates the nodes passed in
- Allows multiple operations and accurate selections within an iteration
  - Allows for deception

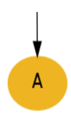
Terminal Primitives

- Random Individuals
  - *count*

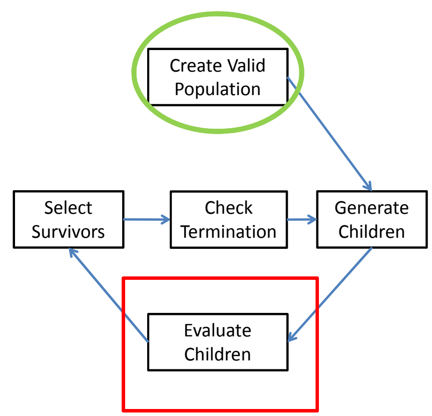


- \`Last' Set

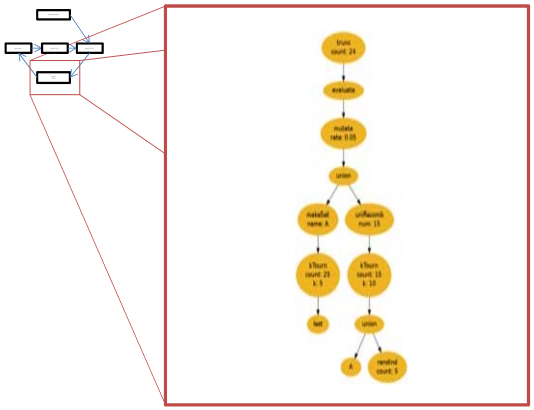
- Persistent Sets
  - *name*



Meta-Genetic Program



BBSA Evaluation



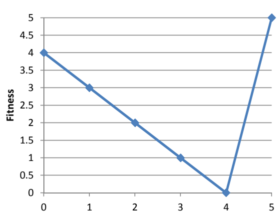
Termination Conditions

- Evaluations
- Iterations
- Operations
- Convergence

Proof of Concept Testing

- Deceptive Trap Problem

0 | 0 | 1 | 1 | 0	0 | 1 | 0 | 1 | 0	1 | 1 | 1 | 1 | 0



Proof of Concept Testing (cont.)

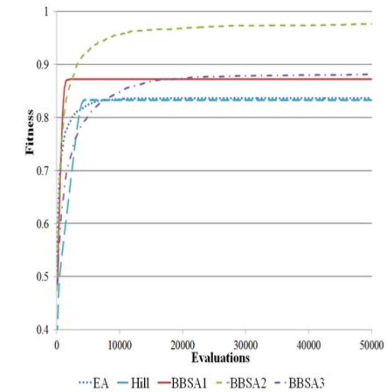
- Evolved Problem Configuration
  - Bit-length = 100
  - Trap Size = 5
- Verification Problem Configurations
  - Bit-length = 100, Trap Size = 5
  - Bit-length = 200, Trap Size = 5
  - Bit-length = 105, Trap Size = 7
  - Bit-length = 210, Trap Size = 7

Results

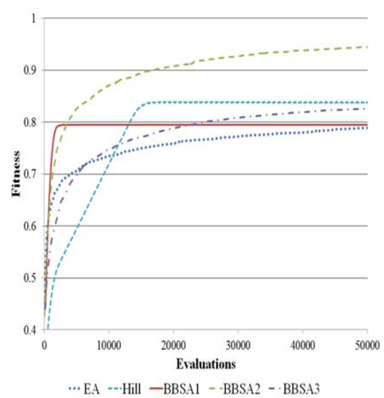
BBSA	EA	Hill-Climber
1	+	+
2	+	+
3	+	+
4	-	-
5	+	+
6	+	+
7	+	+
8	-	-
9	-	-
10	-	-
11	+	+
12	-	-
13	+	+
14	+	+
15	-	-

60% Success Rate

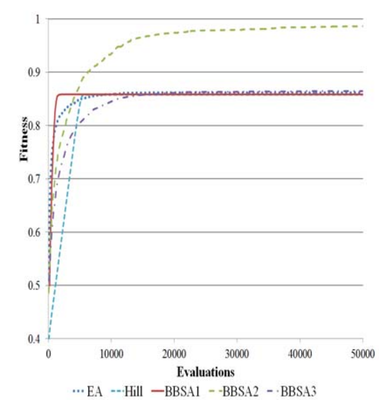
Results: Bit-Length = 100 Trap Size = 5



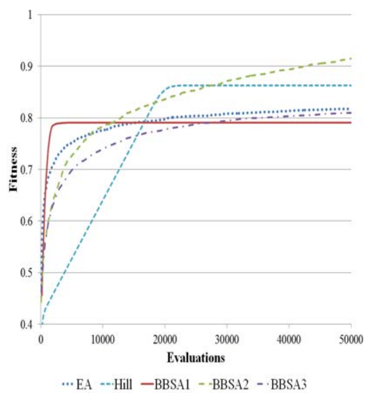
Results: Bit-Length = 200 Trap Size = 5

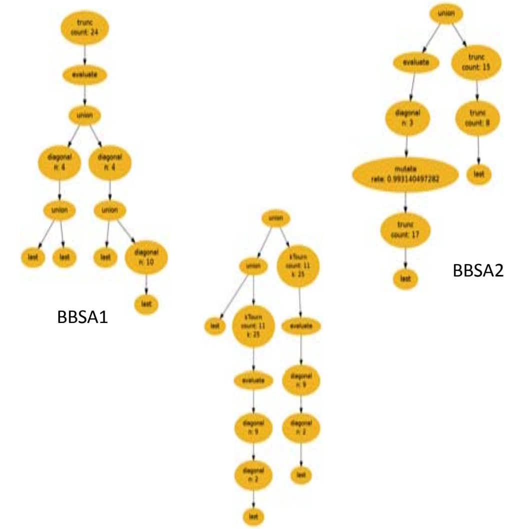


Results: Bit-Length = 105 Trap Size = 7

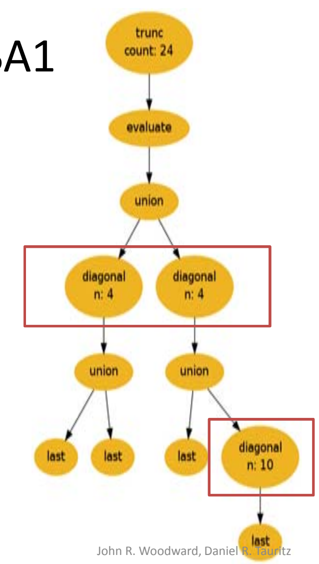


Results:
Bit-Length = 210
Trap Size = 7

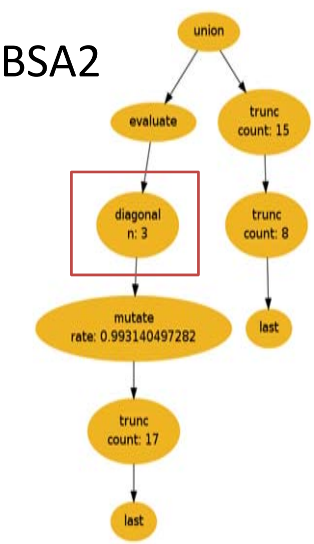




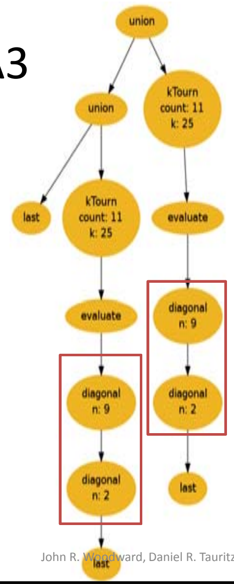
BBSA1



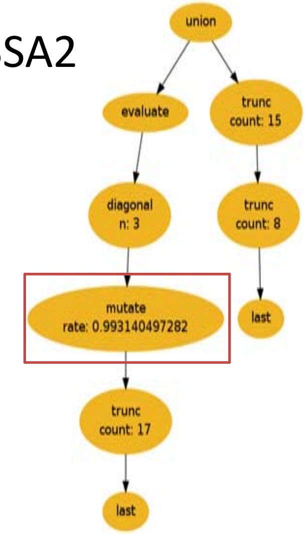
BBSA2



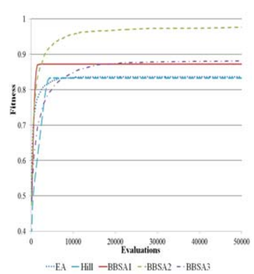
BBSA3



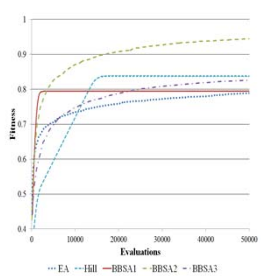
BBSA2



Over-Specialization



Trained Problem Configuration

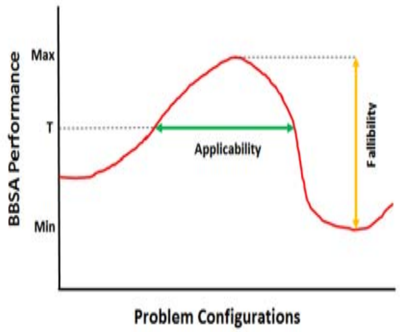


Alternate Problem Configuration

Robustness

- Measures of Robustness
  – Applicability
  – Fallibility
- 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



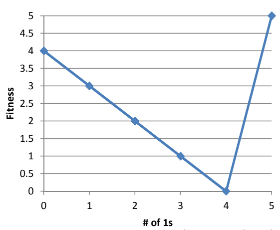
Multi-Sampling

- 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

0 | 0 | 1 | 1 | 0	0 | 1 | 0 | 1 | 0	1 | 1 | 1 | 1 | 0



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

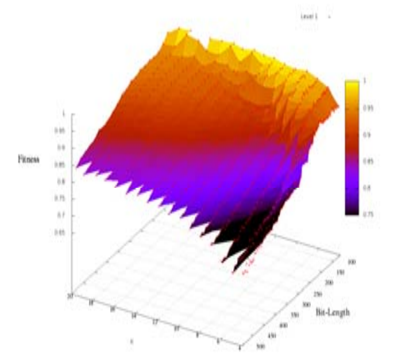
Level	Run	Train Fit.	Test Fit.	Fallibility
1	1	1.0	0.976	0.094
1	2	1.0	0.999	8.33 E-3
1	3	0.944	0.883	0.082
1	4	0.976	0.894	0.224
2	1	0.997	0.996	0.023
2	2	0.992	0.959	0.130
2	3	0.966	0.970	0.054
2	4	0.979	0.947	0.120
3	1	0.965	0.966	0.050
3	2	0.984	0.980	0.065
3	3	0.899	0.886	0.059
3	4	0.926	0.898	0.073
4	1	0.976	0.999	5.00 E-3
4	2	0.973	0.969	.0903
4	3	0.982	0.975	0.059
4	4	0.993	0.999	5.00 E-3
5	1	0.973	0.977	0.050
5	2	0.893	0.879	0.035
5	3	0.850	0.850	0.045
5	4	0.955	0.986	0.029

Level	Run	+	~	−
1	1	11	0	0
1	2	11	0	0
1	3	11	0	0
1	4	6	2	3
2	1	11	0	0
2	2	11	0	0
2	3	11	0	0
2	4	11	0	0
3	1	11	0	0
3	2	11	0	0
3	3	11	0	0
3	4	11	0	0
4	1	11	0	0
4	2	11	0	0
4	3	11	0	0
4	4	11	0	0
5	1	11	0	0
5	2	10	1	0
5	3	7	4	0
5	4	11	0	0

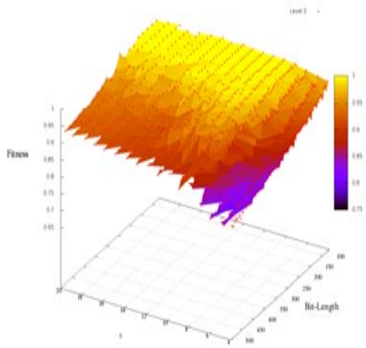
Problem Configuration Landscape Analysis

- Run evolved BBSAs on wider set of problem configurations
- Bit-length: ~75-~500
- Trap Size: 4-20

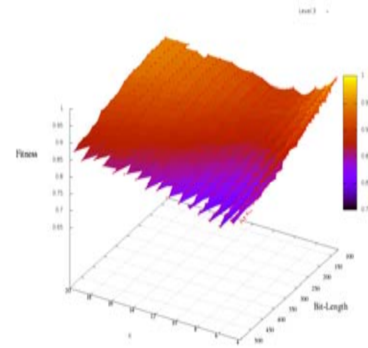
Results: Multi-Sampling Level 1



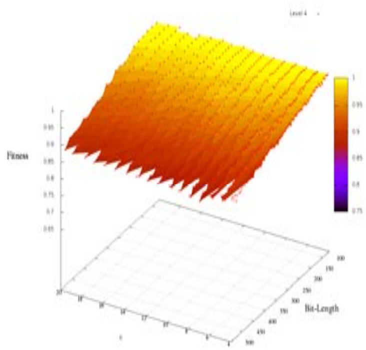
Results: Multi-Sampling Level 2



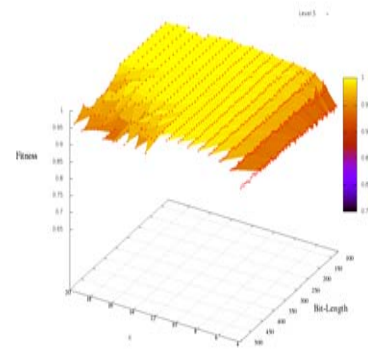
Results: Multi-Sampling Level 3



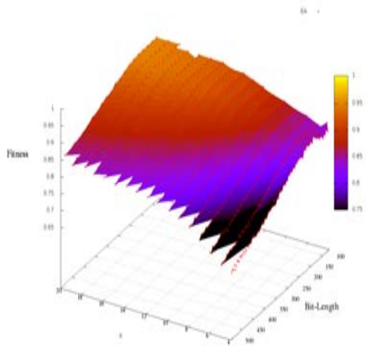
Results: Multi-Sampling Level 4



Results: Multi-Sampling Level 5



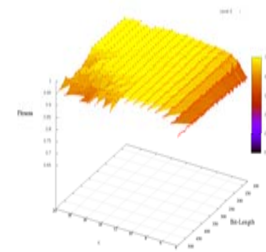
Results: EA Comparison

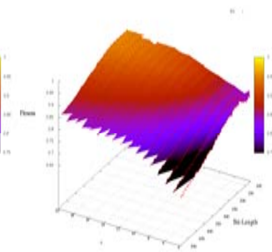


Robustness: Fallibility

Multi-Sample Level 5

Standard EA

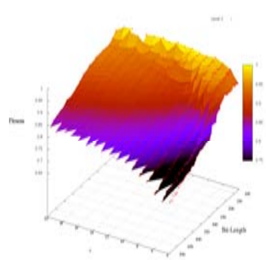


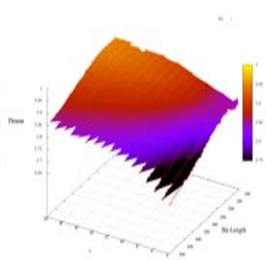


Robustness: Fallibility

Multi-Sample Level 1

Standard EA

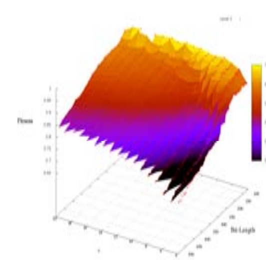


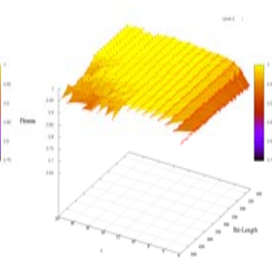


Robustness: Applicability

Multi-Sample Level 1

Multi-Sample Level 5





Robustness: Fallibility

Level	Run	Train Fit.	Test Fit.	Fallibility
5	1	0.973	0.977	0.050
5	2	0.893	0.879	0.035
5	3	0.850	0.850	0.045
5	4	0.955	0.986	0.029

Drawbacks

- Increased computational time
  - More runs per evaluation (increased wall time)
  - More problem configurations to optimize for (increased evaluations)

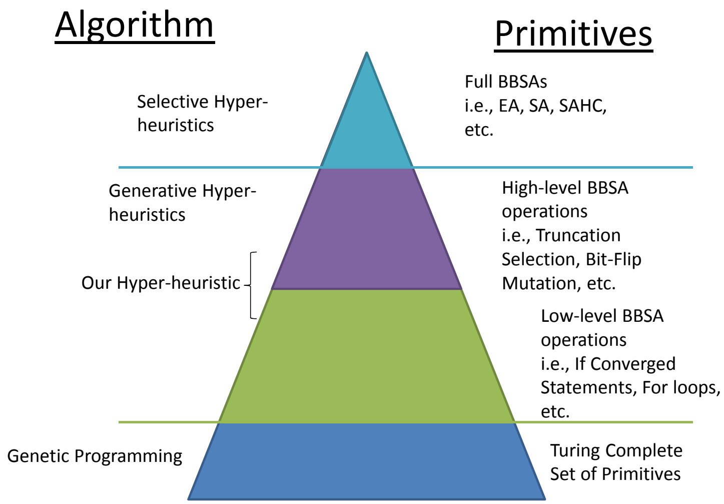
Summary of Multi-Sample Improvements

- Improved Hyper-Heuristic to evolve more robust BBSAs
- Evolved custom BBSA which outperformed standard EA and were robust to changes in problem configuration

Challenges in Hyper-heuristics

- Hyper-heuristics are very computationally expensive (use Asynchronous Parallel GP [26])
- What is the best primitive granularity? (see next slide)
- How to automate decomposition and recomposition of primitives?
- How to automate primitive extraction?
- How does hyper-heuristic performance scale for increasing primitive space size? (see [25])

Primitive Granularity



End of File ☺

- 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 1

1. John Woodward. Computable and Incomputable Search Algorithms and Functions. IEEE International Conference on Intelligent Computing and Intelligent Systems (IEEE ICIS 2009), pages 871-875, Shanghai, China, November 20-22, 2009.
2. John Woodward. The Necessity of Meta Bias in Search Algorithms. International Conference on Computational Intelligence and Software Engineering (CiSE), pages 1-4, Wuhan, China, December 10-12, 2010.
3. John Woodward & Ruibin Bai. Why Evolution is not a Good Paradigm for Program Induction: A Critique of Genetic Programming. In Proceedings of the first ACM/SIGEVO Summit on Genetic and Evolutionary Computation, pages 593-600, Shanghai, China, June 12-14, 2009.
4. Jerry Swan, John Woodward, Ender Ozcan, Graham Kendall, Edmund Burke. Searching the Hyper-heuristic Design Space. Cognitive Computation, 6:66-73, 2014.
5. Gisele L. Pappa, Gabriela Ochoa, Matthew R. Hyde, Alex A. Freitas, John Woodward, Jerry Swan. Contrasting meta-learning and hyper-heuristic research. Genetic Programming and Evolvable Machines, 15:3-35, 2014.
6. Edmund K. Burke, Matthew Hyde, Graham Kendall, and John Woodward. Automating the Packing Heuristic Design Process with Genetic Programming. Evolutionary Computation, 20(1):63-89, 2012.
7. Edmund K. Burke, Matthew R. Hyde, Graham Kendall, and John Woodward. A Genetic Programming Hyper-Heuristic Approach for Evolving Two Dimensional Strip Packing Heuristics. IEEE Transactions on Evolutionary Computation, 14(6):942-958, December 2010.

References 2

8. Edmund K. Burke, Matthew R. Hyde, Graham Kendall, Gabriela Ochoa, Ender Ozcan and John R. Woodward. Exploring Hyper-heuristic Methodologies with Genetic Programming, Computational Intelligence: Collaboration, Fusion and Emergence, In C. Mumford and L. Jain (eds.), Intelligent Systems Reference Library, Springer, pp. 177-201, 2009.
9. Edmund K. Burke, Matthew Hyde, Graham Kendall and John R. Woodward. The Scalability of Evolved On Line Bin Packing Heuristics. In Proceedings of the IEEE Congress on Evolutionary Computation, pages 2530-2537, September 25-28, 2007.
10. R. Poli, John R. Woodward, and Edmund K. Burke. A Histogram-matching Approach to the Evolution of Bin-packing Strategies. In Proceedings of the IEEE Congress on Evolutionary Computation, pages 3500-3507, September 25-28, 2007.
11. Edmund K. Burke, Matthew Hyde, Graham Kendall, and John Woodward. Automatic Heuristic Generation with Genetic Programming: Evolving a Jack-of-all-Trades or a Master of One, In Proceedings of the Genetic and Evolutionary Computation Conference, pages 1559-1565, London, UK, July 2007.
12. John R. Woodward and Jerry Swan. Template Method Hyper-heuristics, Metaheuristic Design Patterns (MetaDeeP) workshop, GECCO Comp'14, pages 1437-1438, Vancouver, Canada, July 12-16, 2014.
13. Saemundur O. Haraldsson and John R. Woodward, Automated Design of Algorithms and Genetic Improvement: Contrast and Commonalities, 4th Workshop on Automatic Design of Algorithms (ECADA), GECCO Comp '14, pages 1373-1380, Vancouver, Canada, July 12-16, 2014.

References 3

14. John R. Woodward, Simon P. Martin and Jerry Swan. Benchmarks That Matter For Genetic Programming, 4th Workshop on Evolutionary Computation for the Automated Design of Algorithms (ECADA), GECCO Comp '14, pages 1397-1404, Vancouver, Canada, July 12-16, 2014.
15. John R. Woodward and Jerry Swan. The Automatic Generation of Mutation Operators for Genetic Algorithms, 2nd Workshop on Evolutionary Computation for the Automated Design of Algorithms (ECADA), GECCO Comp' 12, pages 67-74, Philadelphia, U.S.A., July 7-11, 2012.
16. John R. Woodward and Jerry Swan. Automatically Designing Selection Heuristics. 1st Workshop on Evolutionary Computation for Designing Generic Algorithms, pages 583-590, Dublin, Ireland, 2011.
17. Edmund K. Burke, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Ozcan, and John Woodward. A Classification of Hyper-heuristics Approaches, Handbook of Metaheuristics, pages 449-468, International Series in Operations Research & Management Science, M. Gendreau and J-Y Potvin (Eds.), Springer, 2010.
18. Libin Hong and John Woodward and Jingpeng Li and Ender Ozcan. Automated Design of Probability Distributions as Mutation Operators for Evolutionary Programming Using Genetic Programming. Proceedings of the 16th European Conference on Genetic Programming (EuroGP 2013), volume 7831, pages 85-96, Vienna, Austria, April 3-5, 2013.
19. Ekaterina A. Smorodkina and Daniel R. Tauritz. Toward Automating EA Configuration: the Parent Selection Stage. In Proceedings of CEC 2007 - IEEE Congress on Evolutionary Computation, pages 63-70, Singapore, September 25-28, 2007.

References 4

20. Brian W. Goldman and Daniel R. Tauritz. Self-Configuring Crossover. In Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO '11), pages 575-582, Dublin, Ireland, July 12-16, 2011.
21. Matthew A. Martin and Daniel R. Tauritz. Evolving Black-Box Search Algorithms Employing Genetic Programming. In Proceedings of the 15th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO '13), pages 1497-1504, Amsterdam, The Netherlands, July 6-10, 2013.
22. Nathaniel R. Kamrath, Brian W. Goldman and Daniel R. Tauritz. Using Supportive Coevolution to Evolve Self-Configuring Crossover. In Proceedings of the 15th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO '13), pages 1489-1496, Amsterdam, The Netherlands, July 6-10, 2013.
23. Matthew A. Martin and Daniel R. Tauritz. A Problem Configuration Study of the Robustness of a Black-Box Search Algorithm Hyper-Heuristic. In Proceedings of the 16th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO '14), pages 1389-1396, Vancouver, BC, Canada, July 12-16, 2014.
24. Sean Harris, Travis Bueter, and Daniel R. Tauritz. A Comparison of Genetic Programming Variants for Hyper-Heuristics. In Proceedings of the 17th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO '15), pages 1043-1050, Madrid, Spain, July 11-15, 2015.
25. Matthew A. Martin and Daniel R. Tauritz. Hyper-Heuristics: A Study On Increasing Primitive-Space. In Proceedings of the 17th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO '15), pages 1051-1058, Madrid, Spain, July 11-15, 2015.
26. Alex R. Bertels and Daniel R. Tauritz. Why Asynchronous Parallel Evolution is the Future of Hyper-heuristics: A CDCL SAT Solver Case Study. To Appear in Proceedings of the 18th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO `16), Denver, Colorado, USA, July 2016.