Parallel Problem Solving from Nature

September 13-17, 2014

Ljubljana, Slovenia

Automatic Design of Algorithms via Hyper-heuristic Genetic Programming

John Woodward, Jerry Swan

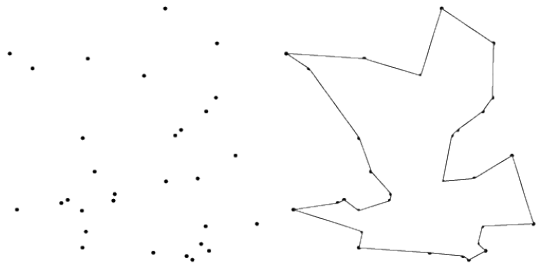
John.Woodward@cs.stir.ac.uk; Jerry.Swan@cs.stir.ac.uk

CHORDS Research Group, Stirling University

http://www.maths.stir.ac.uk/research/groups/chords/

Conceptual Overview

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



Genetic Algorithm
heuristic – permutations

↓

Travelling Salesman

↓

Tour

Single tour NOT EXECUTABLE!!!

Genetic Programming
code fragments in for-loops.

↓

Travelling Salesman Instances

↓

TSP algorithm

EXECUTABLE on MANY INSTANCES!!!

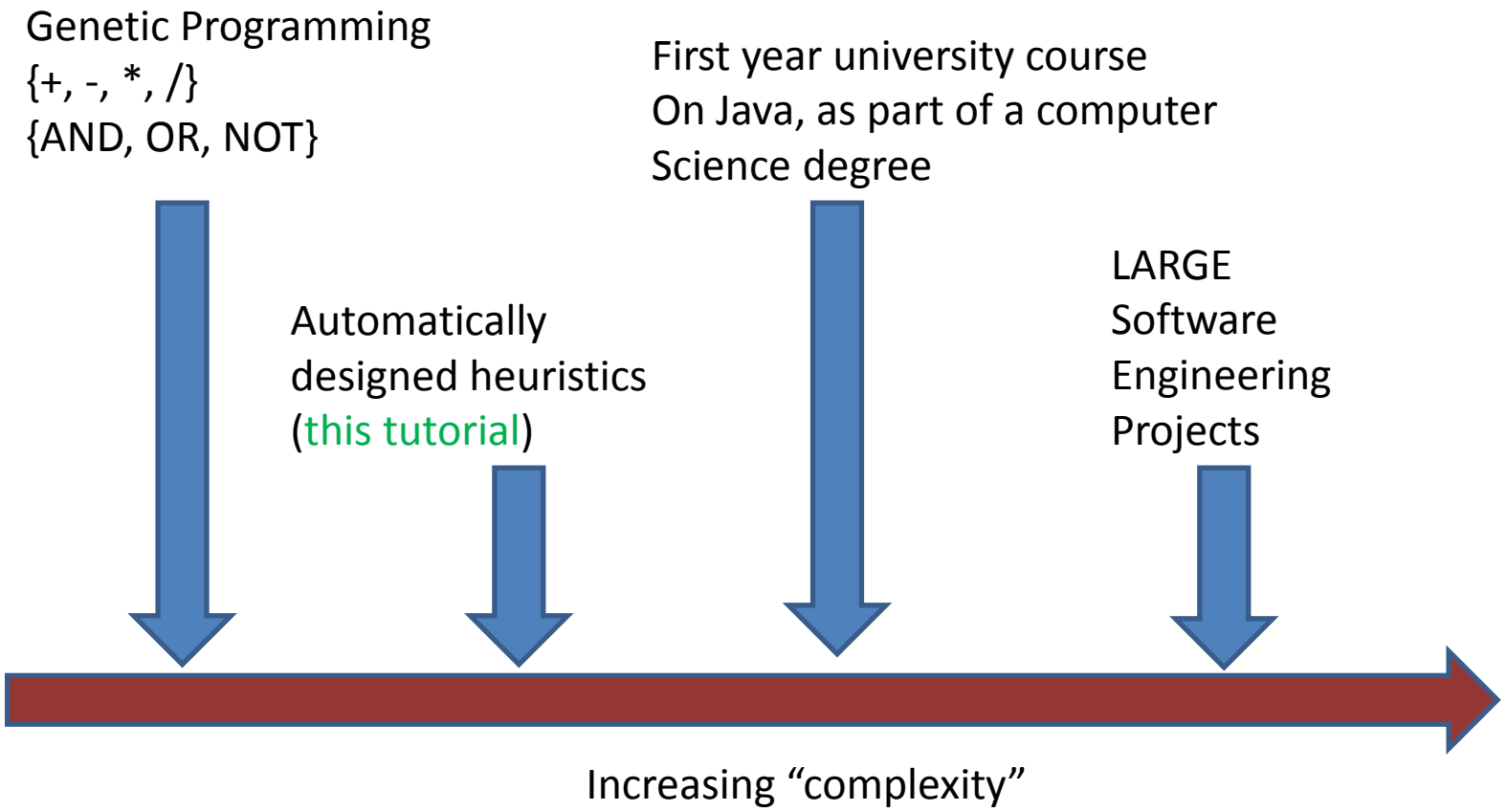
Give a man a fish and he will eat for a **day**.

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

Scalable? General?

New domains for GP

Program Spectrum



Overview of Applications

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

Plan: From Evolution to Automatic Design

1. (assume knowledge of Evolutionary Computation)
2. Evolution, Genetic Algorithms and Genetic Programming (1)
3. Motivations (conceptual and theoretical) (4)
4. Examples of automatic generation

   Genetic Algorithms (selection and mutation) (8)

   Bin packing (9)

   Evolutionary Programming (12)

8. Wrap up. Closing comments (6)

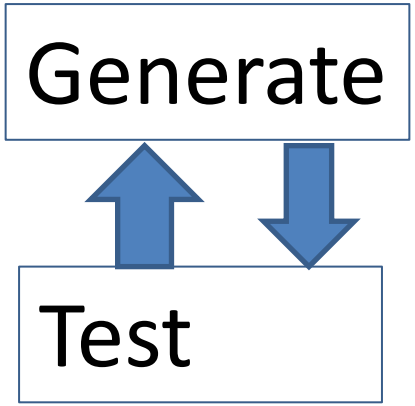
9. Questions (during AND after…) Please ☺

Now is a good time to say you are in the wrong room ☺

Evolution GA/GP

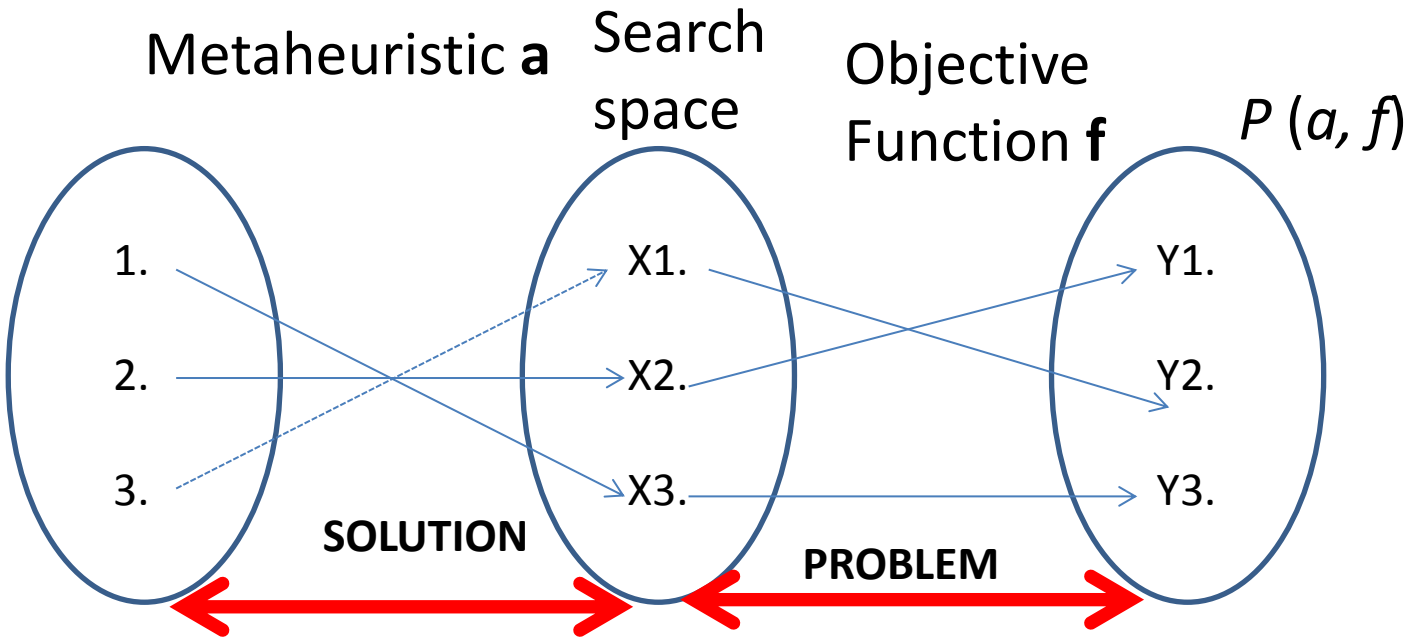
- Generate and test: cars, code, models, proofs, medicine, hypothesis.
- Evolution (select, vary, inherit).
- Fit for purpose

Feedback loop
Humans
Computers



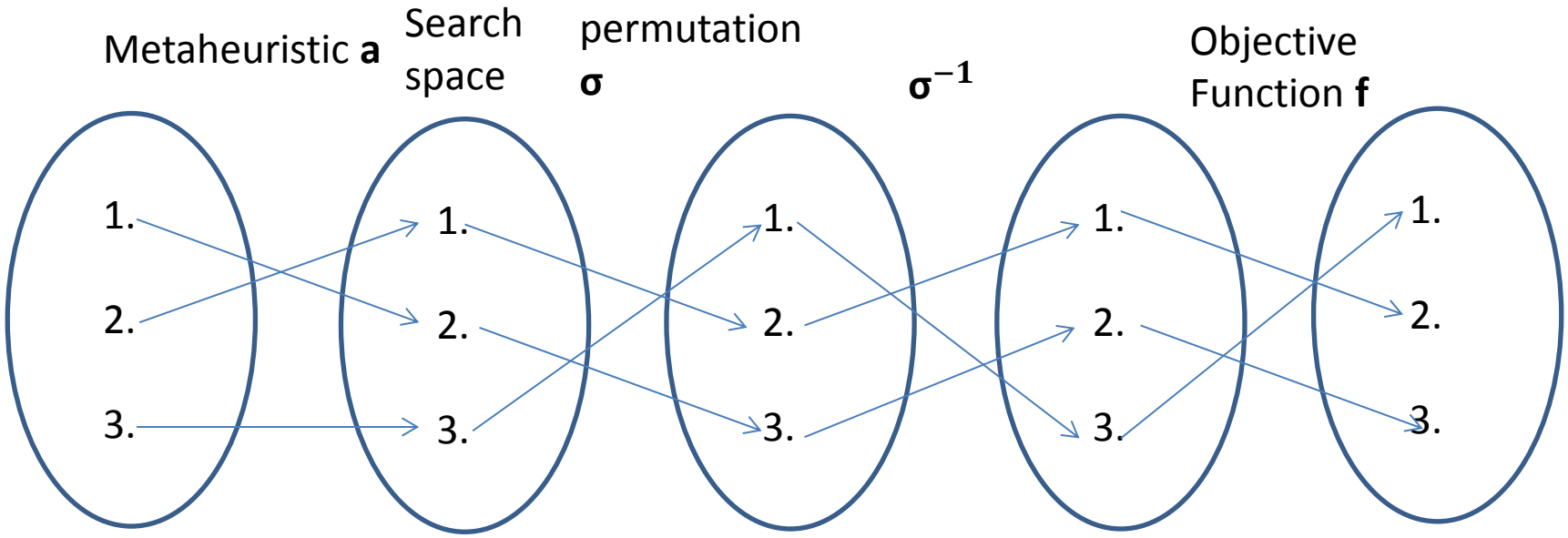
Inheritance
Off-spring have similar Genotype (phenotype)
PERFECT CODE [3]

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 $\boldsymbol{\sigma}\boldsymbol{\sigma}^{-1}\boldsymbol{f}$ (that is $\boldsymbol{f}$)
2. Metaheuristic $\mathbf{a}\boldsymbol{\sigma}$ applied to function $\boldsymbol{\sigma}^{-1}\boldsymbol{f}$ 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.

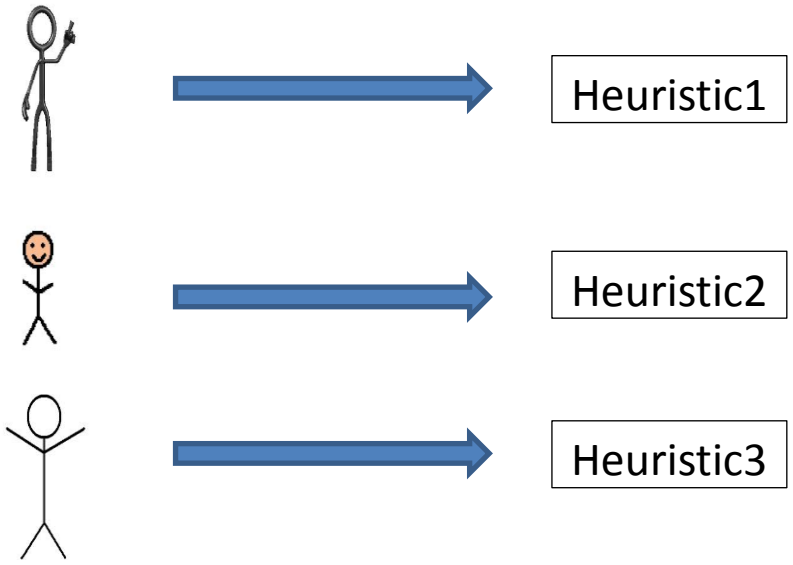
One Man – One/Many Algorithm

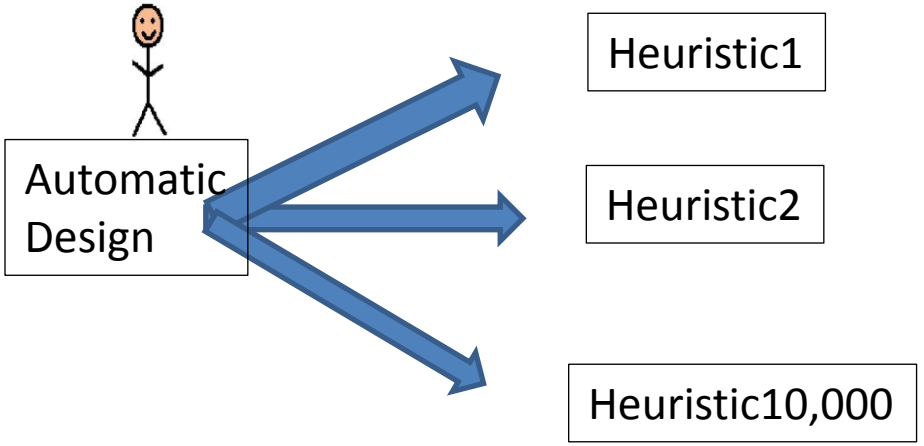
1. Researchers design heuristics by hand and test them on problem instances or arbitrary benchmarks off internet.
2. Presenting results at conferences and publishing in journals. In this talk/paper we propose a new algorithm…

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…





Evolving Selection Heuristics [16]

- **Rank** selection

  P(i) α i

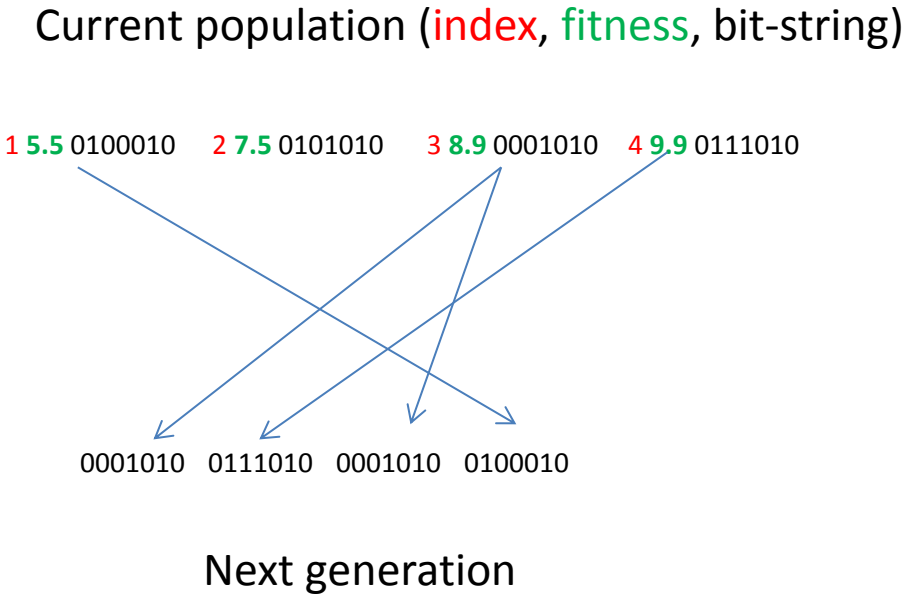
Probability of selection is proportional to the **index** in sorted population

- **Fitness** Proportional

  P(i) α fitness(i)

Probability of selection is proportional to the **fitness**

Fitter individuals are more likely to be selected in both cases.

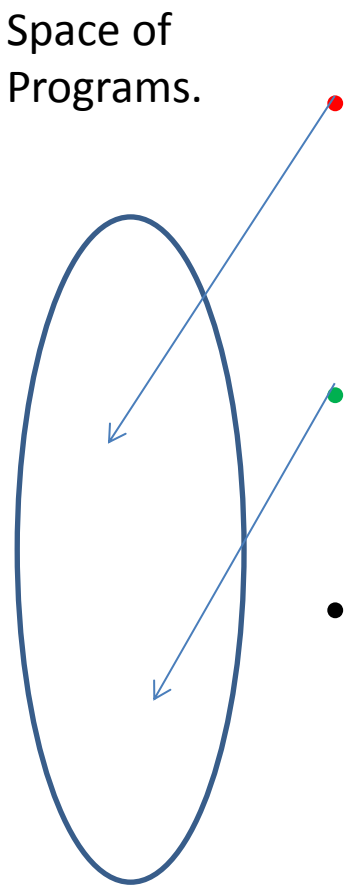


Framework for Selection Heuristics

Selection heuristics operate in the following framework

```
for all individuals p
   in population
select p in
   proportion to
   value( p );
```

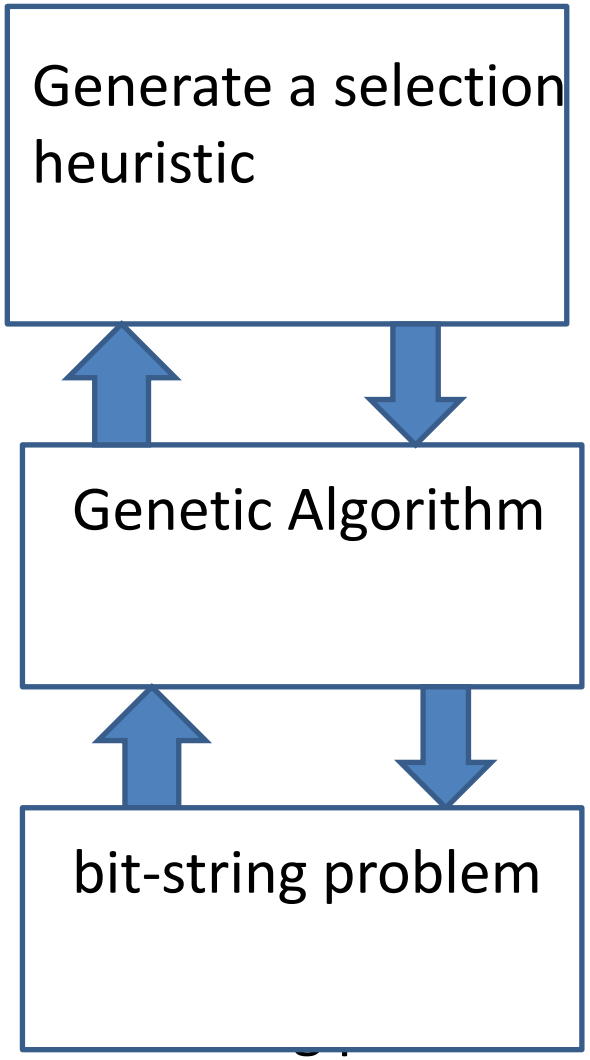
- To perform rank selection replace value with index i.
- To perform fitness proportional selection replace value with fitness



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

Selection Heuristic Evaluation

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



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^{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

	Fitness Proportional	Rank	generated-selector
mean	0.831528	0.907809	**0.916088**
std dev	0.003095	0.002517	**0.006958**
min	0.824375	0.902813	**0.9025**
max	0.838438	0.914688	**0.929063**

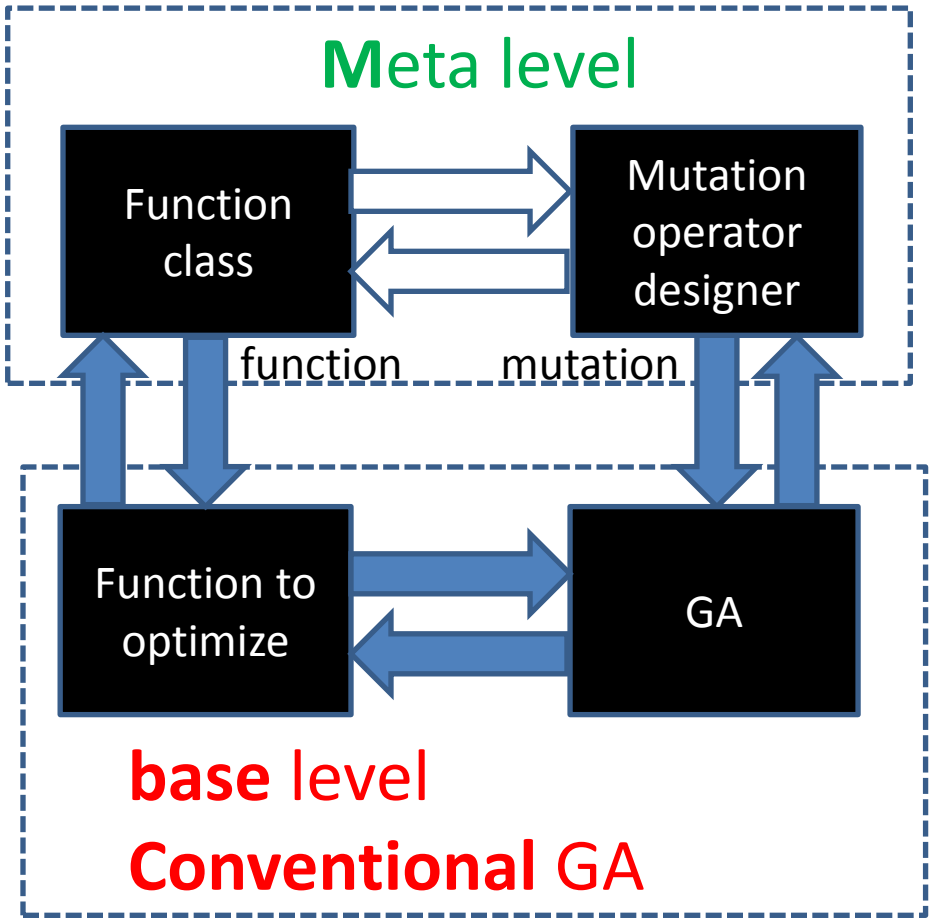
Performing t-test comparisons of fitness-proportional selection and rank selection against generated heuristics resulted in a p-value of better than 10^-*15 in both cases. In both of these* cases the generated heuristics outperform the standard selection operators (rank and fit-proportional).

Take Home Points

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

Meta and Base Learning [15]

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



Compare Signatures (Input-Output)

Genetic Algorithm

- $(B^n \rightarrow R) \rightarrow B^n$

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

Genetic Algorithm FACTORY

- $[(B^n \rightarrow R)] \rightarrow ((B^n \rightarrow R) \rightarrow B^n)$

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

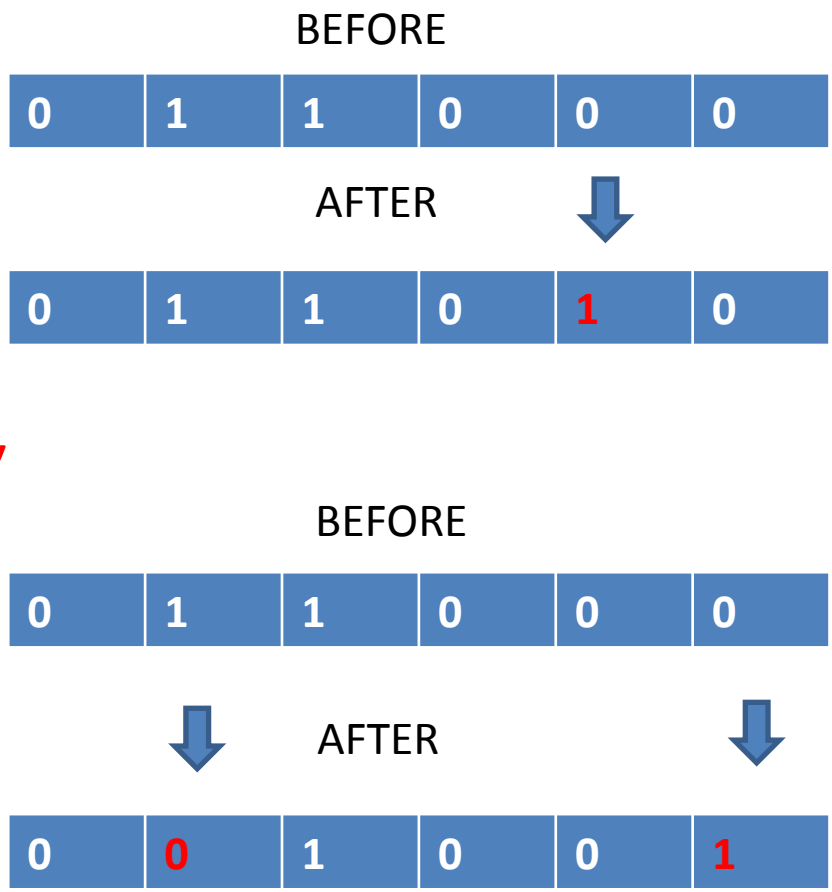
We are **raising the level of generality** at which we operate.

Two Examples of Mutation Operators

- **One point mutation** flips ONE single bit in the genome (bit-string).

(1 point to n point mutation)

- **Uniform mutation** flips ALL bits with a *small probability* p. No matter how we vary p, it will never be one point mutation.
- *Lets invent some more!!!*
- ☹ NO, lets build a general method (for problem class)



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

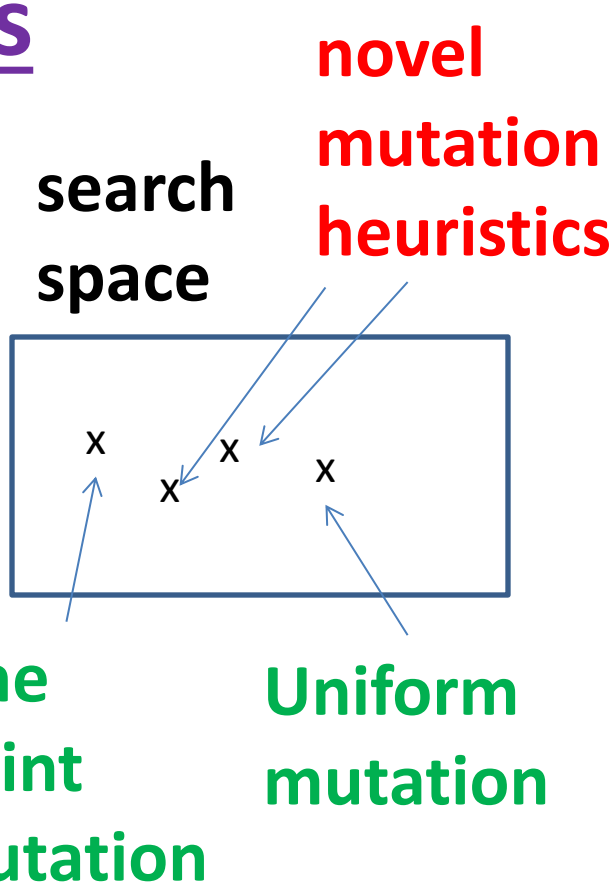
(mutation operators)

Fitness value

Mutation operator

Genetic Algorithm

Base-level



Two search spaces

Commonly used Mutation operators

Building a Space of Mutation Operators

Inc	0
Dec	1
Add	1,2,3
If	4,5,6
Inc	-1
Dec	-2

Program counter pc	2

WORKING REGISTERS

110	-1	+1	43	…	

INPUT-OUTPUT REGISTERS

-20	-1	+1	20	…	

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** Ri ← Rj + Rk
- **Inc** Ri ← Ri + 1
- **Dec** Ri ← Ri − 1
- **Ivt** Ri ← −1 ∗ Ri
- **Clr** Ri ← 0
- **Rnd** Ri ← Random([−1, +1]) //mutation rate
- **Set** Ri ← 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

Line	UNIFORM	ONE POINT MUTATION
0	**Rpt, 33, 18**	**Rpt, 33, 18**
1	Nop	Nop
2	Nop	Nop
3	Nop	Nop
4	**Inc, 3**	**Inc, 3**
5	Nop	Nop
6	Nop	Nop
7	Nop	Nop
8	**IfRand, 3, 6**	**IfRand, 3, 6**
9	Nop	Nop
10	Nop	Nop
11	Nop	Nop
12	**Ivt,–3**	**Ivt,–3**
13	Nop	**Stp**
14	Nop	Nop
15	Nop	Nop
16	Nop	Nop

- **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.

number	**function**
1	x
2	sin2(x/4 − 16)
3	(x − 4) ∗ (x − 12)
4	(x ∗ x − 10 ∗ cos(x))
5	sin(pi∗x/64−4) ∗ cos(pi∗x/64−12)
6	sin(pi∗cos(pi∗x/64 − 12)/4)
7	1/(1 + x /64)

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^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,numbits]). This value is then **fed into one of the 7 functions.**

Results – 32 bit problems

Problem classes Means and standard deviations	Uniform Mutation	One-point mutation	generated-mutation
p1 mean	30.82	30.96	**31.11**
p1 std-dev	0.17	0.14	0.16
p2 mean	951	959.7	**984.9**
p2 std-dev	9.3	10.7	10.8
p3 mean	506.7	512.2	**528.9**
p3 std-dev	7.5	6.2	6.4
p4 mean	945.8	954.9	**978**
p4 std-dev	8.1	8.1	7.2
p5 mean	0.262	0.26	**0.298**
p5 std-dev	0.009	0.013	0.012
p6 mean	0.432	0.434	**0.462**
p6 std-dev	0.006	0.006	0.004
p7 mean	0.889	0.89	**0.901**
p7 std-dev	0.002	0.003	0.002

Results – 64 bit problems

Problem classes Means and stand dev	Uniform Mutation	One-point mutation	generated-mutation
p1 mean	55.31	56.08	**56.47**
p1 std-dev	0.33	0.29	0.33
p2 mean	3064	3141	**3168**
p2 std-dev	33	35	33
p3 mean	2229	2294	**2314**
p3 std-dev	31	28	27
p4 mean	3065	3130	**3193**
p4 std-dev	36	24	28
p5 mean	0.839	0.846	**0.861**
p5 std-dev	0.012	0.01	0.012
p6 mean	0.643	0.643	**0.663**
p6 std-dev	0.004	0.004	0.003
p7 mean	0.752	0.7529	**0.7684**
p7 std-dev	0.0028	0.004	0.0031

p-values T Test for 32 and 64-bit functions on the7 problem classes

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

Rebuttal to Reviews

1. Did we test the new mutation operators against standard operators (one-point and uniform mutation) on **different problem classes**?

- NO – the mutation operator is designed (evolved) specifically for that class of problem.

2. Are we taking the **training stage** into account?

- NO, we are just comparing mutation operators in the testing phase – Anyway how could we meaningfully compare “brain power” (manual design) against “processor power” (evolution).

3. Train for all functions – NO, we are specializing.

Additions to Genetic Programming

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

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

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

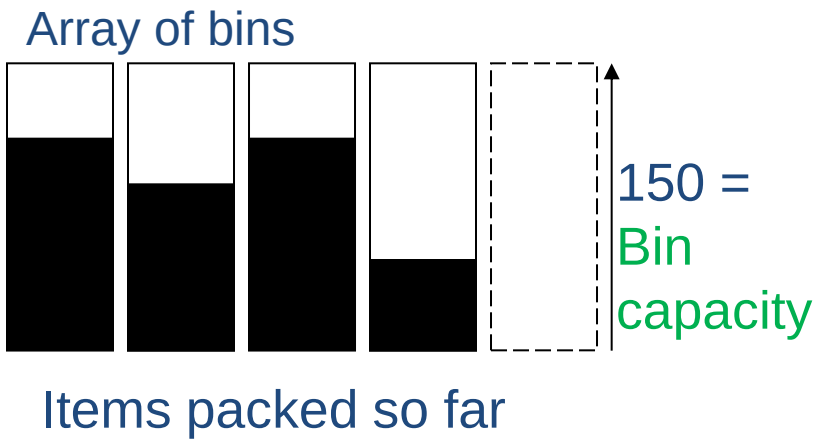
NOW OVER TO JERRY SWAN

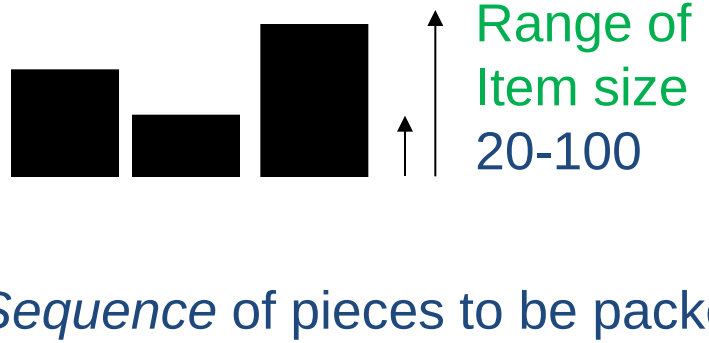
Problem Classes Do Occur

1. Problem classes are probability distributions over problem instances.
2. **Travelling Salesman**
    1. Distribution of cities over different counties
    2. E.g. USA is square, Japan is long and narrow.
3. **Bin Packing & Knapsack Problem**
    1. The items are drawn from some probability distribution.
4. Problem classes do occur in the real-world
5. Next slides demonstrate **problem classes** and **scalability** with on-line bin packing.

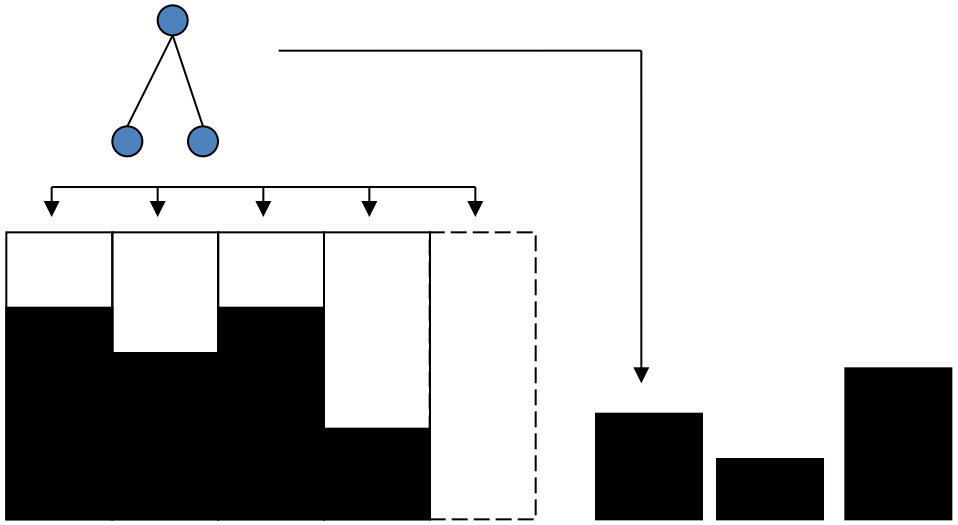
On-line Bin Packing Problem [9,11]

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

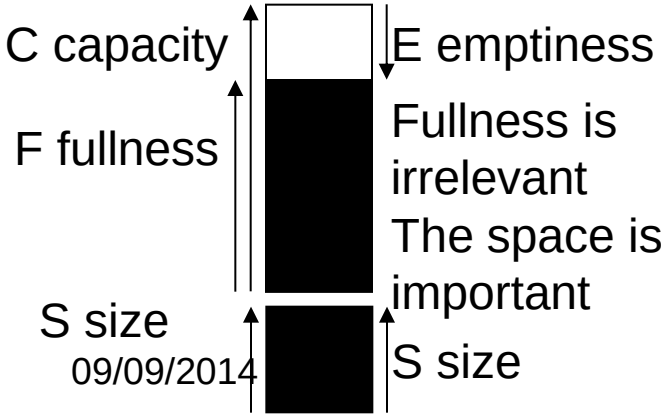




Genetic Programming applied to on-line bin packing

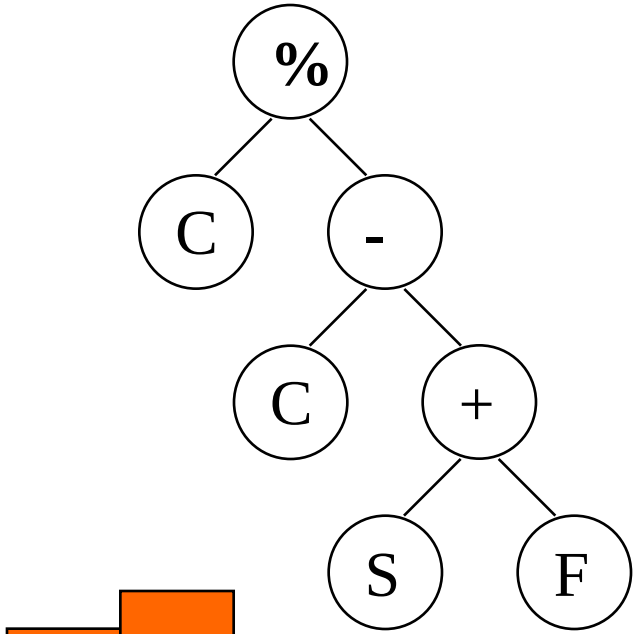


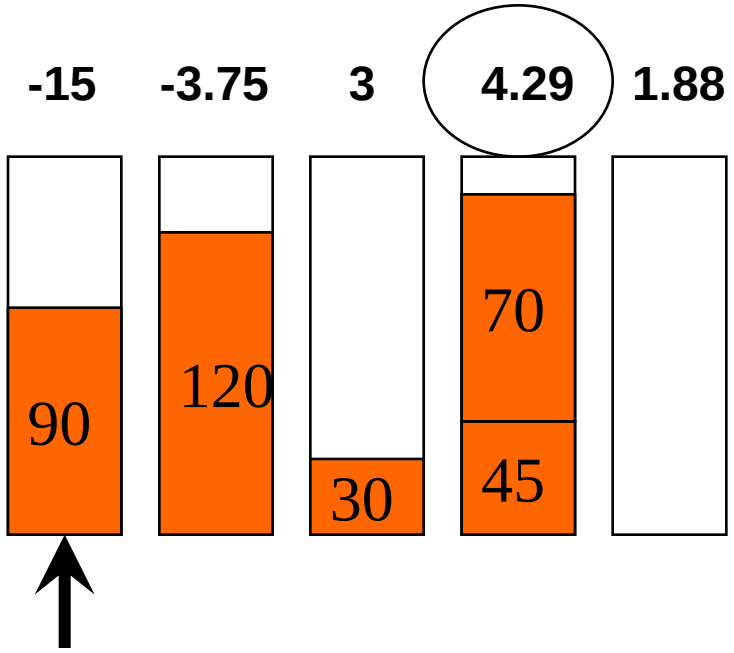
Not obvious how to link Genetic Programming to combinatorial problems.
The GP tree is applied to each bin with the current item and placed in the bin with
The maximum score



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

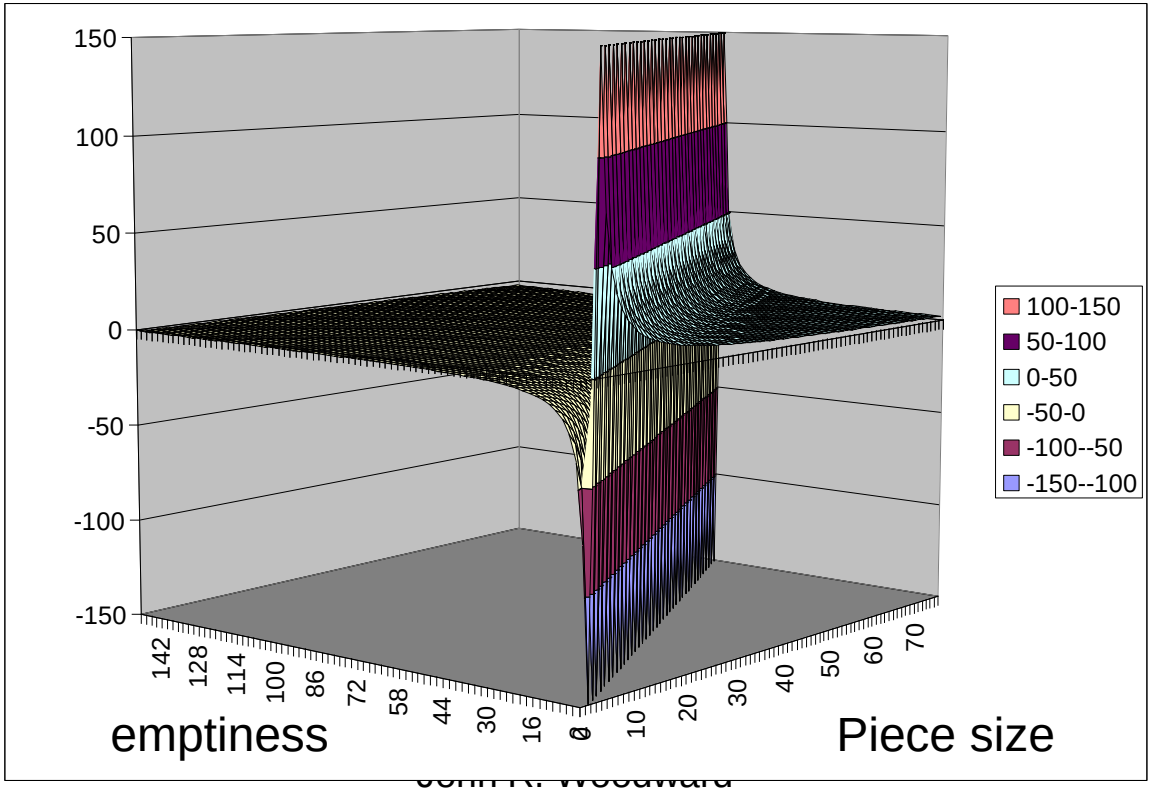
How the heuristics are applied (skip)



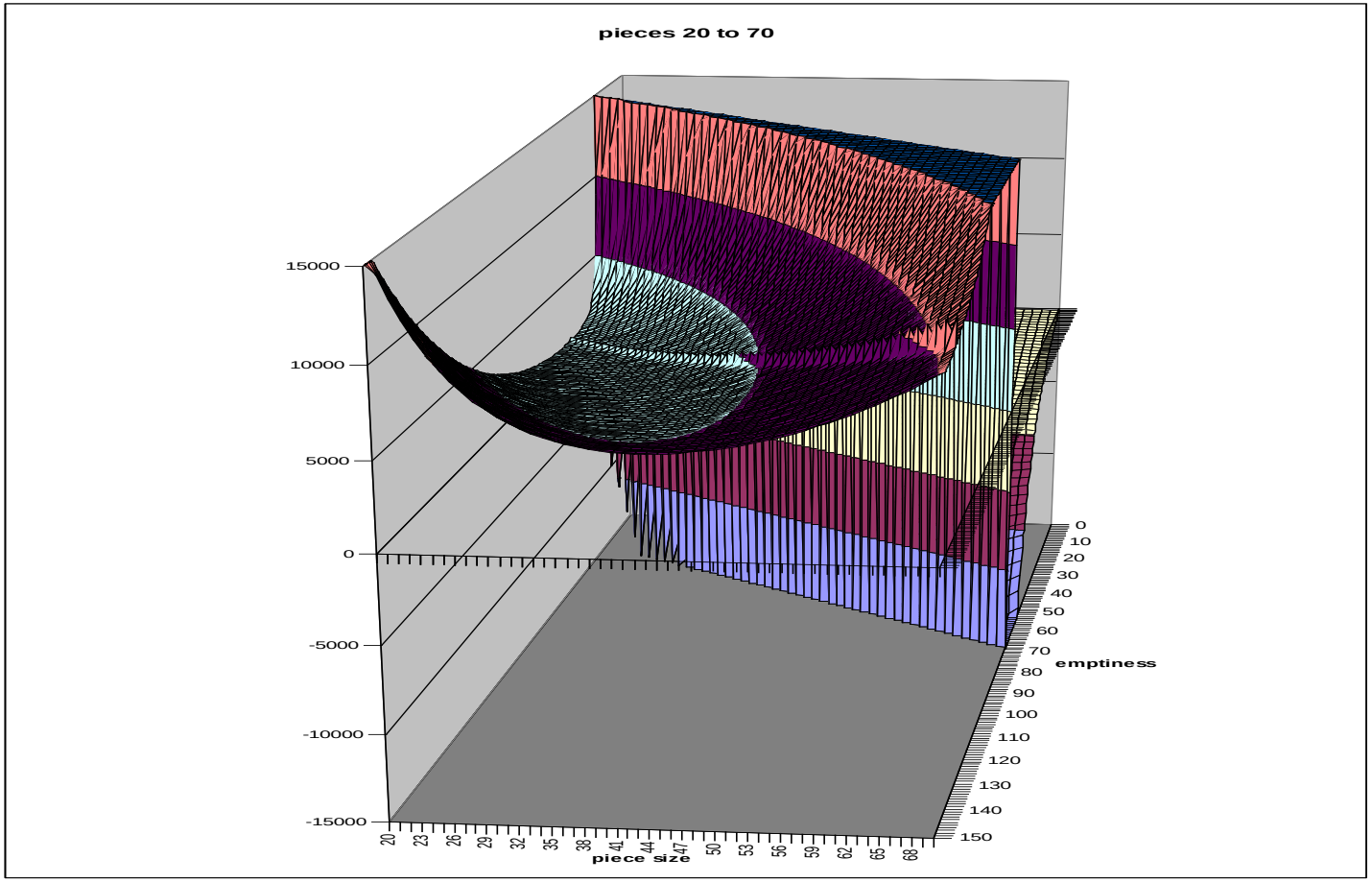


The Best Fit Heuristic

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.

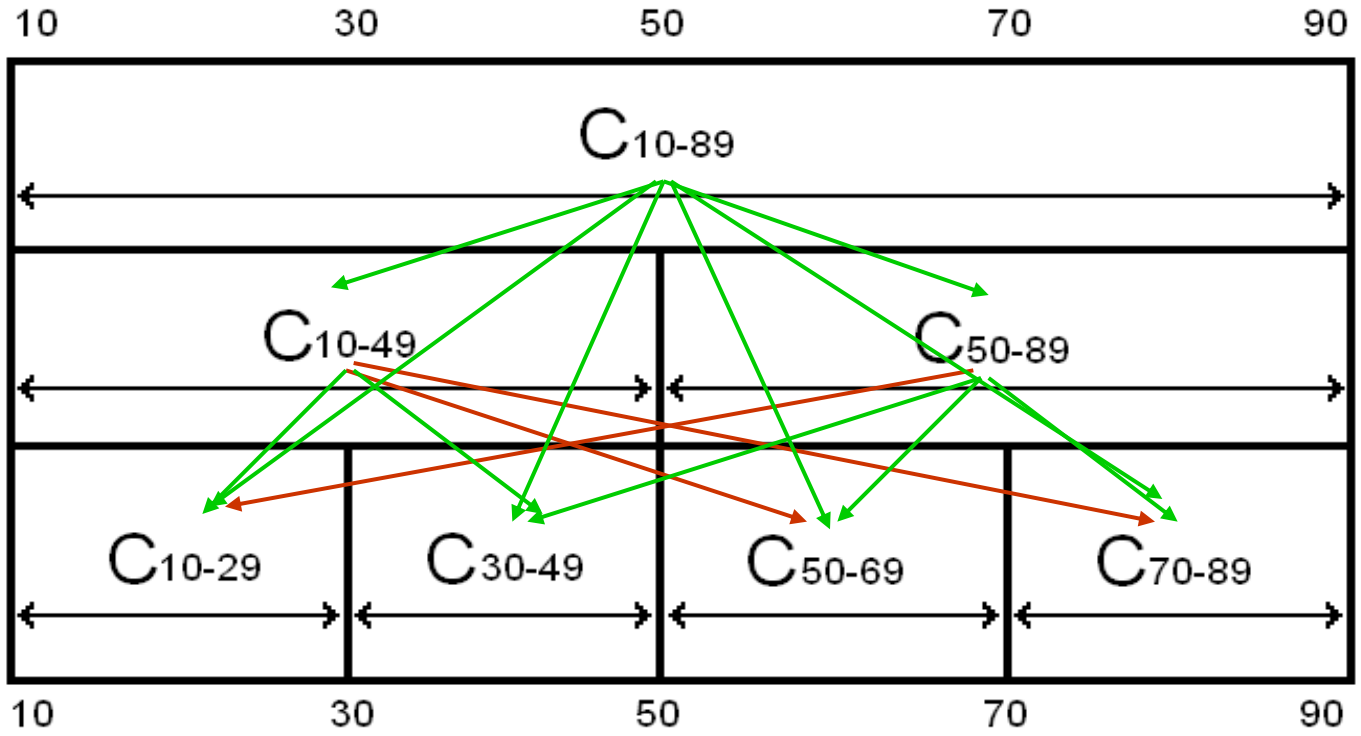


Similar shape to best fit – but curls up in one corner.
Note that this is rotated, relative to previous slide.

Robustness of Heuristics

= all legal results

= some illegal results



Testing Heuristics on problems of much larger size than in training

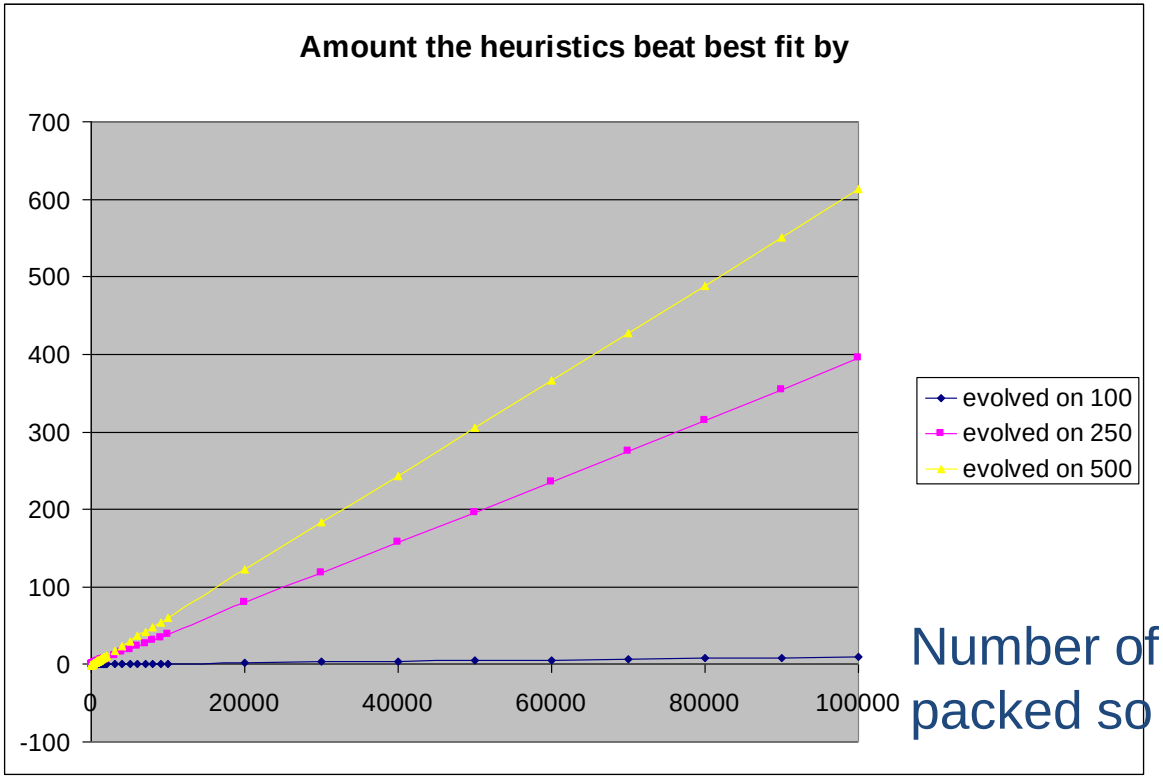
Table I	*H trained100*	*H trained 250*	*H trained 500*
100	0.427768358	0.298749035	0.140986023
1000	0.406790534	0.010006408	0.000350265
10000	0.454063071	2.58E-07	9.65E-12
100000	0.271828318	1.38E-25	2.78E-32

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.

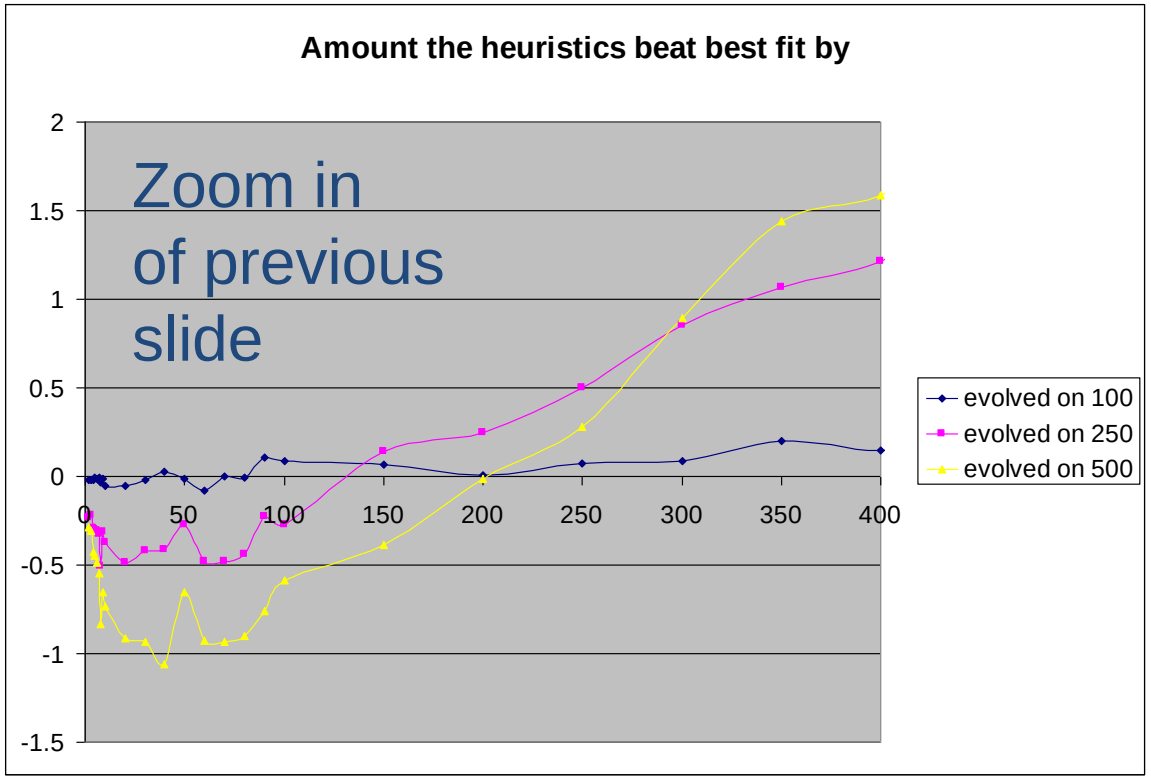


Number of pieces packed so far.

- 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.

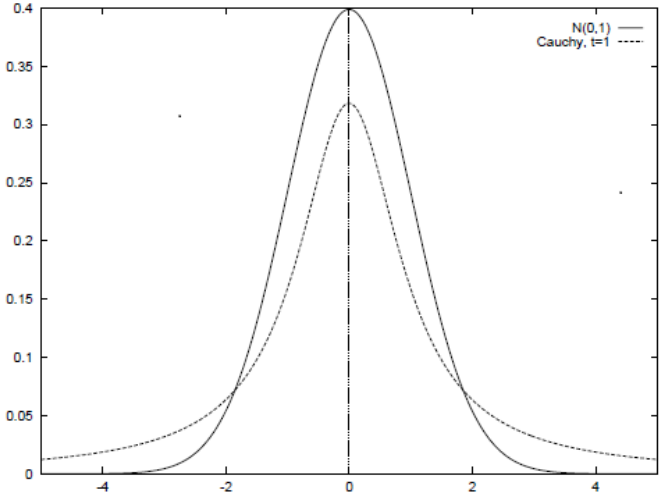


- The heuristic seems to learn the number of pieces in the problem
- Analogy with sprinters running a race – accelerate towards end of race.
- The "break even point" is approximately half of the size of the training problem size
- If there is a gap of size 30 and a piece of size 20, it would be better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

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** (a probability distributions over functions).
6. NO CROSSOVER

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



Genotype is (1.2,...,4.4,…,8.6) After mutation

(Fast) Evolutionary Programming

Heart of algorithm is mutation
SO LETS AUTOMATICALLY DESIGN

$$x_i'(j) = x_i(j) + \eta_i(j) D_j$$

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

1. Generate the initial population of 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
$f_7(x) = \sum_{i=1}^{n} i x_i^4 + random[0, 1)$	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

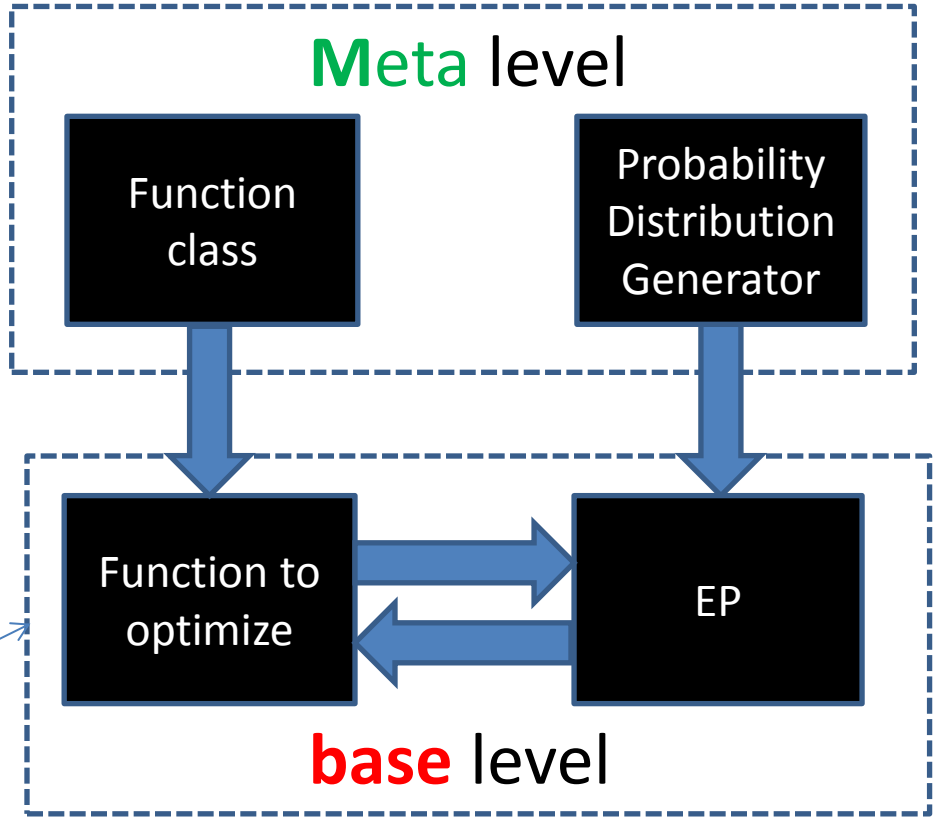
1. Machine learning needs to generalize.
2. We generalize to function classes.
3. y = x^2 (**a function**)
4. y = ax^2(parameterised function)
5. y = ax^2, a ~[1,2] (**function class**)
6. We do this for all benchmark functions.
7. **The mutation operators is evolved to fit the probability distribution 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
$f_2(x) = a\sum_{i=1}^{n} \mid x_i \mid + b\prod_{i=1}^{n} \mid x_i \mid$	$[-10, 10]^n$	$b \in [0, 10^{-5}]$	0
$f_3(x) = \sum_{i=1}^{n}(a\sum_{j=1}^{i} x_j)^2$	$[-100, 100]^n$	N/A	0
$f_4(x) = \max_i\{a \mid x_i \mid, 1 \leq i \leq n\}$	$[-100, 100]^n$	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
$f_8(x) = \sum_{i=1}^{n} -(x_i \sin(\sqrt{\lvert x_i \rvert}) + a)$	$[-500, 500]^n$	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

$(R^n \rightarrow R) \rightarrow R^n$

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.

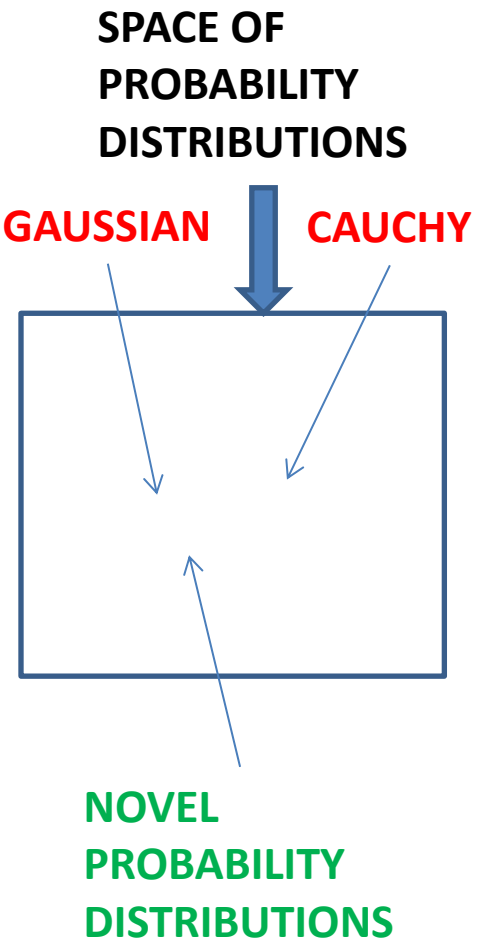
Genetic Programming to Generate Probability Distributions

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.

	ADR1	ADR2	ADR3	ADR4	ADR5	ADR6	ADR7	ADR8	ADR9	ADR10	ADR11	ADR12	ADR13	ADR14	ADR15	ADR16	ADR17	ADR18	ADR19	ADR20	ADR21	ADR22	ADR23
f_1	**3.796042378**	3.796795988	3.796172495	3.800097053	3.798215201	3.798426888	3.79682023	2531.207552	3.796057571	**3.795990923**	3.845998009	3.796186664	3.796277155	3.796649368	3.79612197	1655.307735	16670.67898	3.802819106	4.260445107	3.796097541	44.18750067	3.796066481	3.816827509
	(15.53395359)	(15.53405343)	(15.53391881)	(15.53360785)	(15.53375201)	(15.53375598)	(15.53387687)	(11290.25839)	(15.53388934)	(15.53393945)	(15.52984118)	(15.533912)	(15.53392309)	(15.53392525)	(15.53399083)	(2395.677635)	(7873.980089)	(15.53117437)	(15.5103041)	(15.53398613)	(49.93392927)	(15.5339502)	(15.53473388)
f_2	0.03272533	**0.017265507**	0.06411854	0.243765938	0.227029932	0.242415867	0.141152255	10.89831484	**0.012640918**	**0.015574102**	0.878284278	0.048279715	0.068163286	0.10166228	0.033010725	5.652569538	62.08781058	0.02943591	**0.003081819**	**0.008143766**	0.269907184	0.033088529	0.039637186
	(0.006220059)	(0.00316433)	(0.012214985)	(0.052833925)	(0.041526229)	(0.045421712)	(0.027254796)	(29.61351836)	(0.002366678)	(0.003232023)	(0.224644483)	(0.009029109)	(0.013313298)	(0.021204407)	(0.006539326)	(6.56695685)	(15.77341379)	(0.005428489)	(0.000845787)	(0.001531136)	(0.148966146)	(0.006487538)	(0.038535293)
f_3	0.059115953	0.542598852	**0.018626247**	0.103132914	**0.013948627**	**0.014236266**	**0.005922877**	33.77718802	2.665366951	2.039338209	0.937932626	0.28659926	0.064134902	0.097019896	0.084879154	3433.708363	7087.024237	0.157436096	50.23471658	5.582568993	11.01739701	0.199225997	182.1465227
	(0.127613356)	(0.907149879)	(0.038845313)	(0.052212998)	(0.006159356)	(0.005987797)	(0.003058048)	(25.60578768)	(3.027285258)	(2.680462701)	(0.471271529)	(0.420720664)	(0.100019319)	(0.113519449)	(0.123340191)	(3872.306081)	(4298.4011)	(0.194641319)	(47.12256504)	(6.896844468)	(7.777401427)	(0.380877388)	(157.1263872)
f_4	17.25966091	19.41813374	16.15346434	**0.292628803**	9.648461377	14.04875432	16.79279446	28.13359508	10.99661065	14.19501939	**0.162317701**	2.630226876	4.399393447	1.330615099	21.57790808	68.40219553	74.66207074	40.88032128	37.07012251	16.70289914	61.04798152	15.80925637	8.929701211
	(4.966599608)	(7.253429181)	(4.887330993)	(0.502787856)	(5.399453121)	(7.007590006)	(7.260923198)	(43.09163947)	(5.899934832)	(6.765307096)	(0.125141891)	(2.042641555)	(2.852710021)	(1.700571261)	(7.946864347)	(7.994013124)	(7.257141818)	(9.606000218)	(14.00520825)	(5.748158881)	(8.858525787)	(4.977945429)	(5.603363277)
f_5	-12.59873403	-11.39294662	**-13.38616242**	-12.17766112	**-12.98617719**	**-13.26385156**	**-13.4343626**	4522608.395	-12.80651566	-12.38416526	-10.21081863	-12.61795619	-12.07327907	-12.34060394	-11.37346075	768.5563656	189434.0968	-11.6837873	-10.20070261	-10.91325651	6.018008713	-12.79281207	-7.353638881
	(19.08466205)	(16.77875418)	(19.29489759)	(20.17833659)	(18.94371583)	(19.30900217)	(19.83357655)	(4559249.522)	(19.38895811)	(17.82800791)	(18.25643711)	(18.513)	(17.92596386)	(17.0817524)	(18.02353828)	(1491.144115)	(135909.7748)	(18.82095681)	(17.37621129)	(18.05348947)	(25.80280349)	(19.30159697)	(20.10064511)
f_6	422.0335	1143.0035	12.165	0.135	0.058	**0.0535**	0.11	9.298	0.107	**0.015**	0.3255	**0.039**	**0.035**	**0.0315**	195.777	17987.881	37431.6995	424.4385	375.2155	**0.017**	4999.8915	185.43	83.996
	(1649.46925)	(2074.280164)	(38.34787739)	(0.264724286)	(0.221943568)	(0.222858818)	(0.304613647)	(6.862988915)	(0.25034923)	(0.022826577)	(0.157228463)	(0.061034934)	(0.066451407)	(0.024553915)	(490.1220371)	(13480.82092)	(19447.51813)	(752.3238003)	(888.0006481)	(0.025975697)	(5268.862537)	(446.2366362)	(123.8021751)
f_7	0.065542671	0.078308367	0.056194615	0.08411716	**0.047030838**	0.04882212	**0.048746647**	2.807243879	0.063532032	0.058626761	0.194904423	0.061920648	0.053685635	0.062253766	0.070244942	0.532824769	45.5222079	0.091505903	0.0869747	0.06586313	0.478628757	0.06288902	0.186500061
	(0.010787905)	(0.024299068)	(0.011619042)	(0.019571033)	(0.00605618)	(0.00653602)	(0.0077357)	(1.21388786)	(0.01280476 5)	(0.009280782)	(0.041805987)	(0.010715646)	(0.008060171)	(0.010746513)	(0.010506748)	(0.461115144)	(19.50162907)	(0.022831626)	(0.025469636)	(0.012332483)	(0.175096278)	(0.015746188)	(0.111182366)
f_8	-7476.802093	-7873.512071	-8371.08948	-11531.07729	-8638.189529	-8567.792279	-8461.852799	**-12471.54499**	-10836.75693	-10864.24065	-11921.10551	-11261.25295	-11017.45819	-11363.51973	-7712.376423	-9103.974424	-7289.483153	-7716.478481	-11715.11843	-10802.04284	-7922.383673	-8271.935456	-12347.43418
	(648.3654776)	(503.1138893)	(779.6316972)	(307.0584479)	(533.7011029)	(489.8047705)	(682.1430256)	(158.4399895)	(498.9742452)	(528.4915592)	(319.6841114)	(332.9818897)	(352.2752319)	(396.6847647)	(673.4167468)	(629.0672603)	(588.0128451)	(519.3255199)	(392.4030967)	(383.8291898)	(534.7702565)	(686.2946167)	(208.9940377)
f_9	-6.364340621	-6.613834318	-7.274192496	-8.841127079	-6.525185945	-6.624821944	-6.117765239	-7.490475004	**-8.85403752**	-8.854016567	-8.724342579	-8.853598499	-8.853174849	-8.852121206	-5.713104239	-8.427859381	11.80045883	-6.427611891	-8.754991403	**-8.85405211**	-5.865365514	-7.252578007	-8.832262084
	(13.55211805)	(14.20272875)	(13.10504988)	(11.27787199)	(13.91130284)	(14.11686839)	(14.36718746)	(11.30142186)	(11.28014913)	(11.28014763)	(11.2529754)	(11.28005901)	(11.27996138)	(11.2798762)	(15.24928375)	(11.22098317)	(19.66542025)	(13.47596148)	(11.37032461)	(11.28015264)	(13.78071024)	(13.42757703)	(11.25921778)
f_{10}	-26.54875358	-27.62462039	-27.87344964	-28.52972749	-28.43776411	-27.47491154	-28.12468594	-7.253656564	**-28.58283002**	**-28.58195607**	-23.84876736	-28.57454432	-28.57126049	-28.56345724	-27.30007959	-16.86690042	-4.344066607	-26.58932429	-28.48336407	**-28.58356645**	-23.42001988	-27.41309828	-28.54008083
	(8.642935672)	(7.694296506)	(7.111158515)	(6.28071026 3)	(6.447779575)	(8.117435238)	(6.629475271)	(14.70586537)	(6.292607345)	(6.292379922)	(13.2608428)	(6.290797959)	(6.290015625)	(6.288416006)	(7.779335696)	(9.531070088)	(2.227440058)	(8.952959967)	(6.385591493)	(6.292680027)	(9.536175616)	(7.761631077)	(6.286327759)
f_{11}	-0.651086092	-0.430374859	-0.699398783	-0.732192039	-0.696833115	-0.722930189	-0.681421101	-0.715900311	-0.640236618	-0.674184211	**-0.738435445**	-0.717184562	-0.735309936	-0.725127919	-0.619405568	72.67346703	245.0964286	-0.378051882	-0.315772447	-0.585609275	5.001410223	-0.595487053	-0.732208916
	(0.553088246)	(0.589562784)	(0.533991987)	(0.551795141)	(0.565782865)	(0.543780683)	(0.546113958)	(0.545831558)	(0.558785685)	(0.546230838)	(0.548435173)	(0.564381489)	(0.544806443)	(0.54116403)	(0.534518486)	(74.07522252)	(108.7039575)	(0.519124016)	(0.641591441)	(0.539005548)	(4.930986743)	(0.539010124)	(0.548139483)
f_{12}	2.375745683	1.519931686	0.81756969	0.000049864	1.138872739	1.081912616	2.613131902	295302505.2	0.086323699	0.048961716	0.000761494	**0.000002742**	0.023901023	0.000009886	1.502825411	34.73265169	15022643.14	2.147365455	0.200806538	0.121246677	18.16207189	1.133574956	42557.38789
	(2.850960856)	(2.154711219)	(1.297295505)	(1.32181E-05)	(1.204159184)	(2.165644097)	(3.63485628)	(562235782.9)	(0.29953258)	(0.176719921)	(0.000235795)	(1.09208E-06)	(0.059797341)	(3.87911E-06)	(1.325423866)	(35.78526991)	(24577491.03)	(1.758359464)	(0.423450793)	(0.321512405)	(10.05733127)	(1.058239746)	(190322.4214)
f_{13}	7.528102918	11.54428374	5.380220018	0.000668631	3.178046205	5.343480627	10.81477842	451382341.6	0.003800314	0.001868601	0.010121716	**4.50175E-05**	**6.34555E-05**	0.00012633	3.661952338	333.0155182	7358392.064	18.23218457	0.715447776	0.028620163	257.2859935	5.854254043	527.9210413
	(11.50730323)	(12.5903624)	(10.27590186)	(0.000138084)	(5.035811692)	(5.584041018)	(19.74462848)	(611339756.7)	(0.007130406)	(0.004563481)	(0.002309003)	(2.01955E-05)	(1.65034E-05)	(2.92279E-05)	(4.589088775)	(304.2916941)	(4794487.129)	(21.70208849)	(1.571904151)	(0.071766746)	(118.5776707)	(7.478665087)	(2359.940113)
f_{14}	1.734470721	1.772821551	1.230172967	1.072006527	1.693127216	1.806118977	1.165167477	**0.754781615**	1.292947098	1.048015299	**0.780822999**	**0.833866789**	1.242364004	**0.838551326**	1.463164304	2.630454628	0.99349036	1.348920833	1.250299666	1.651843912	1.535512059	1.045701707	0.871834862
	(1.392570373)	(1.516523557)	(0.555507449)	(0.659369932)	(1.661752265)	(1.345479048)	(0.596149785)	(0.17253071)	(1.07692978)	(0.572821944)	(0.174823651)	(0.309901545)	(1.224806407)	(0.344005191)	(0.857558242)	(3.711827934)	(0.547260035)	(0.870950738)	(1.992292006)	(1.303367715)	(1.417919709)	(0.683565682)	(0.346950096)
f_{15}	**0.000643324**	0.001546119	0.000784565	0.000765787	0.002250898	0.001921141	0.001197892	0.001639261	**0.000535974**	**0.000417704**	0.001101053	**0.000580249**	0.000580289	**0.00063437**	**0.00070720 1**	0.001545556	0.001225589	0.001411983	**0.000496184**	**0.000441549**	**0.000656427**	**0.000479392**	0.000871313
	(0.000543038)	(0.004444802)	(0.00086432)	(0.000855092)	(0.0058185)	(0.003532418)	(0.001824797)	(0.001952972)	(0.000372644)	(0.000285176)	(0.001179448)	(0.000400097)	(0.000362559)	(0.000385607)	(0.000543025)	(0.004456308)	(6.87867E-06)	(0.003855699)	(0.000386388)	(0.000345109)	(0.00043209)	(0.000307943)	(0.0004467)
f_{16}	-1.92447213	-1.924473263	-1.924470875	-1.924468207	-1.924439114	-1.924429001	-1.924455664	-1.924280681	-1.924473516	-1.924473529	-1.924454275	-1.924473348	-1.924472904	-1.924472224	-1.924472113	**-1.924473543**	-1.92447336	-1.924472505	-1.924473543	-1.924473528	-1.924473215	-1.924473016	-1.92447352
	(0.579460448)	(0.579460606)	(0.579460481)	(0.579458449)	(0.579459593)	(0.579458044)	(0.579458793)	(0.57965821)	(0.579460635)	(0.579460628)	(0.579457791)	(0.579460625)	(0.579460516)	(0.579460528)	(0.579460642)	(0.579460632)	(0.579460629)	(0.57946053)	(0.579460632)	(0.579460632)	(0.579460613)	(0.579460543)	(0.579460633)
f_{17}	3.786952682	3.786952414	3.786953089	3.786953821	3.786960153	3.786965855	3.786956682	3.787007025	**3.786952351**	**3.786952347**	3.78696211	3.786952403	3.78695256	3.786952596	3.786952688	**3.786952342**	**3.786952395**	3.78695256	3.786953207	**3.786952346**	3.786952426	3.786952433	3.786952348
	(2.374931079)	(2.374931155)	(2.374931035)	(2.374930466)	(2.374929682)	(2.374928026)	(2.374930239)	(2.374940369)	(2.374931165)	(2.374931168)	(2.374931608)	(2.374931147)	(2.374931129)	(2.374931132)	(2.374931083)	(2.374931169)	(2.374931161)	(2.374931106)	(2.374933216)	(2.374931167)	(2.374931153)	(2.374931141)	(2.374931167)
f_{18}	4.575031533	**4.574971819**	4.575098811	4.575282035	4.5763558	4.577539016	4.575858015	4.588715133	**4.574958404**	**4.574958205**	4.576371722	**4.574967813**	4.575001205	4.575011001	4.575028689	6.005761337	**4.574969199**	**4.575008958**	**4.57495705**	**4.574957813**	**4.574974777**	**4.574979692**	**4.57495876**
	(1.117668123)	(1.117658005)	(1.117699009)	(1.117651066)	(1.117796611)	(1.118208418)	(1.117843077)	(1.112380559)	(1.117656673)	(1.117656611)	(1.118018727)	(1.11765998)	(1.1176575)	(1.117655702)	(1.117663585)	(6.14484409)	(1.117659974)	(1.117664001)	(1.11765655)	(1.117656608)	(1.117661509)	(1.117662402)	(1.117656655)
f_{19}	-3.552067105	-3.552080968	-3.552050408	-3.551996944	-3.551673503	-3.538670434	-3.551882996	-3.55009538	-3.552083616	-3.552083593	-3.551749707	-3.55208093	-3.552074315	-3.552064298	-3.552065788	**-3.552084117**	-3.552080969	-3.552072265	**-3.552084001**	-3.552083898	-3.552079811	-3.552078513	-3.552083673
	(1.903081977)	(1.903089133)	(1.903075519)	(1.903055044)	(1.902886076)	(1.915377418)	(1.903020205)	(1.901950912)	(1.903090389)	(1.903090534)	(1.902932525)	(1.903089578)	(1.903087517)	(1.903082855)	(1.903081611)	(1.903090722)	(1.903089041)	(1.903085633)	(1.903090608)	(1.903090605)	(1.903088929)	(1.903089015)	(1.903090478)

Step by Step Guide to Automatic Design of Algorithms [8, 12]

1. Study the literature for existing heuristics for your chosen domain (manually designed heuristics).
2. Build an algorithmic framework or template which expresses the known heuristics (todo ref)
3. Let Genetic Programming supply variations on the theme.
4. Train and test on problem instances drawn from the same probability distribution (like machine learning). Constructing an optimizer is machine learning (this approach prevents “cheating”).

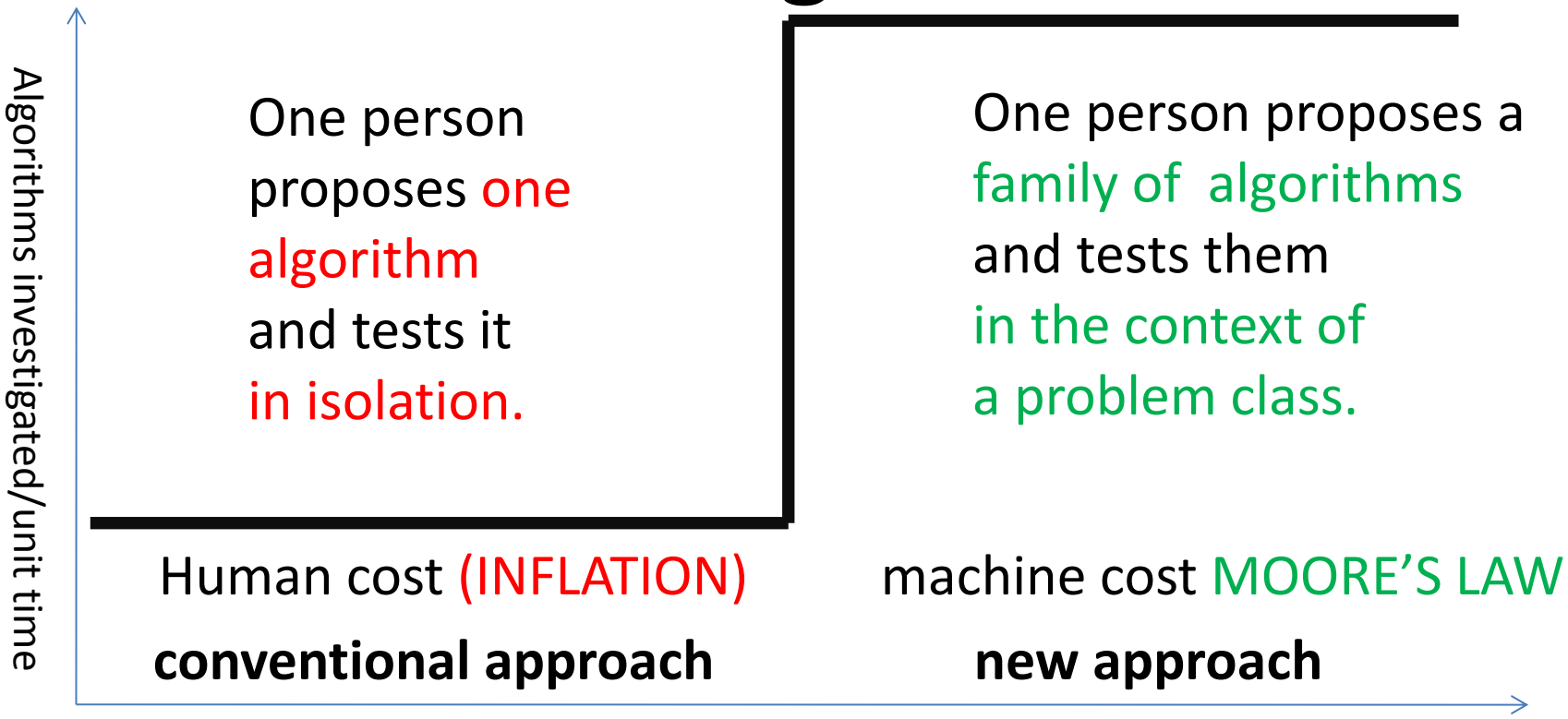
A Brief History (Example Applications) [5]

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

Comparison of Search Spaces

- If we tackle a problem instance directly, e.g. Travelling Salesman Problem, we get a combinatorial explosion. The search space consists of *solutions,* and therefore explodes as we tackle larger problems.
- If we tackle a generalization of the problem, we do not get an explosion as the distribution of functions expressed in the search space tends to a limiting distribution. The search space consists of *algorithms to 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).

Overview of Applications

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

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.

End of File ☺

- Thank you for listening
- I am glad to take any
  - comments (+,-)
  - Criticisms

  Please email me any references.

  http://www.cs.stir.ac.uk/~jrw/

References 1

1. Woodward J. Computable and Incomputable Search Algorithms and Functions. IEEE International Conference on Intelligent Computing and Intelligent Systems (IEEE ICIS 2009) November 20-22,2009 Shanghai, China.

2. John Woodward. The Necessity of Meta Bias in Search Algorithms. International Conference on Computational Intelligence and Software Engineering 2010. CiSE 2010.

3. Woodward, J. & Bai, R. (2009) Why Evolution is not a Good Paradigm for Program Induction; A Critique of Genetic Programming 2009 World Summit on Genetic and Evolutionary Computation (2009 GEC Summit) June 12-14 Shanghai, China

4. J. Swan, J. Woodward, E. Ozcan, G. Kendall, E. Burke. “Searching the Hyper-heuristic Design Space,” Cognitive Computation

5. G.L. Pappa, G. Ochoa, M.R. Hyde, A.A. Freitas, J. Woodward, J. Swan “Contrasting meta-learning and hyper-heuristic research” Genetic Programming and Evolvable Machines.

6. Burke E. K., Hyde M., Kendall G., and Woodward J. 2011. Automating the Packing Heuristic Design Process with Genetic Programming. Evolutionary Computation

References 2

7. Burke, E. K. and Hyde, M. R. and Kendall, G. and Woodward, J., A Genetic Programming Hyper-Heuristic Approach for Evolving Two Dimensional Strip Packing Heuristics, IEEE Transactions on Evolutionary Computation, 2010.

8. E. K. Burke, M. R. Hyde, G. Kendall, G. Ochoa, E. Ozcan and J. R. Woodward (2009) 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

9. Burke E. K., Hyde M., Kendall G., and Woodward J. R. Scalability of Evolved On Line Bin Packing Heuristics Proceedings of Congress on Evolutionary Computation 2007 September 2007

10. Poli R., Woodward J. R., and Burke E. K. A Histogram-matching Approach to the Evolution of Bin-packing Strategies Proceedings of Congress on Evolutionary Computation 2007 September 2007.

11. Burke E. K., Hyde M., Kendall G., and Woodward J. Automatic Heuristic Generation with Genetic Programming: Evolving a Jack-of-all-Trades or a Master of One Proceedings of Genetic and Evolutionary Computation Conference 2007 London UK.

12. John Woodward and Jerry Swan, Template Method Hyper-heuristics Metaheuristic Design Patterns (MetaDeeP) GECCO 2014, Vancouver.

References 3

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 GECCO 2014, Vancouver.

14. John R. Woodward, Simon P. Martin and Jerry Swan, Benchmarks That Matter For Genetic Programming, 4th Workshop on Automatic Design of Algorithms GECCO 2014, Vancouver.

15. J. Woodward and J. Swan, "The Automatic Generation of Mutation Operators for Genetic Algorithms" in 2nd Workshop on Evolutionary Computation for designing Generic Algorithms, GECCO 2012, Philadelphia. DOI: http://dx.doi.org/10.1145/2330784.2330796.

16. John Robert Woodward and Jerry Swan. Automatically designing selection heuristics. In GECCO 2011 1st workshop on evolutionary computation for designing generic algorithms pages 583-590, Dublin, Ireland, 2011.

17. E. K. Burke, M. Hyde, G. Kendall, G. Ochoa, E. Ozcan, and J. Woodward (2009). A Classification of Hyper-heuristics Approaches, Handbook of Metaheuristics, International Series in Operations Research & Management Science, M. Gendreau and J-Y Potvin (Eds.), Springer.

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,