GECCO 1st workshop on Evolving Generic Algorithms.

Automatically Designing Selection Heuristics

John Woodward The University of Nottingham, China
john.woodward@nottingham.edu.cn
Jerry Swan
School of Computer Science, University of Nottingham,
jerry.swan@nottingham.ac.uk

Outline of talk

- Generate and test – fit for a purpose.
  - Generate and test generate and test methods.
- No Free Lunch, problem instances and problem classes.
- Generic Algorithm =

  Genetic Programming + Application Framework
- Selection Heuristics (rank and fitness proportional). Two instances of a more general setting.
- Experiments + Results.

Generate and Test Approach

A "solution" is generated.

It is tested on a problem instance

Each solution is assigned a score (real value).

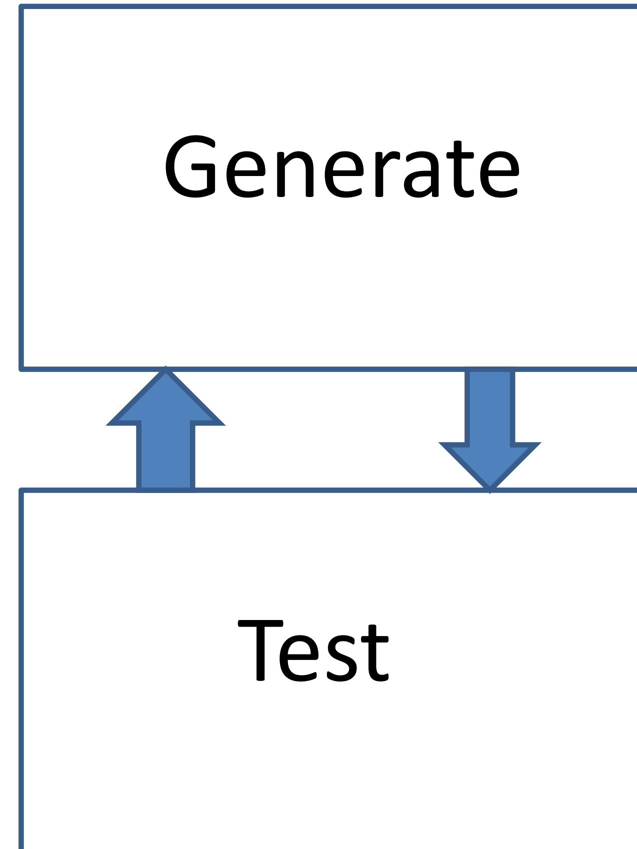
This process is repeated until

time expires or a solution is found.

Includes much of machine learning.

We try to improve the quality of solutions generated by using feedback in this loop.

Alternative but equivalent view is we are sampling a space.

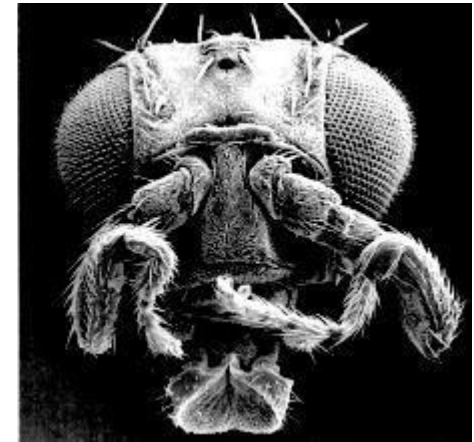


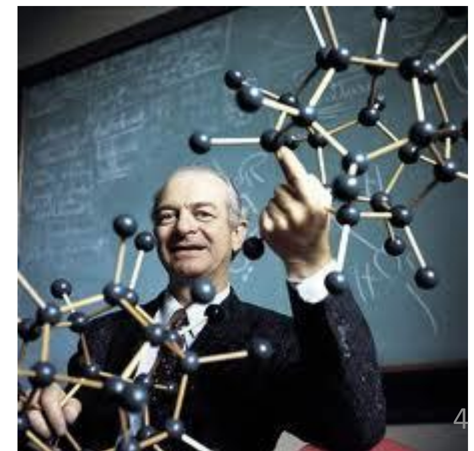
Generate and Test Examples

- Manufacture e.g. cars
- Evolution “survival of the fittest”
- “The best way to have a good

idea is to have lots of ideas” (Linus Pauling).

- Computer code is also generate and tested.





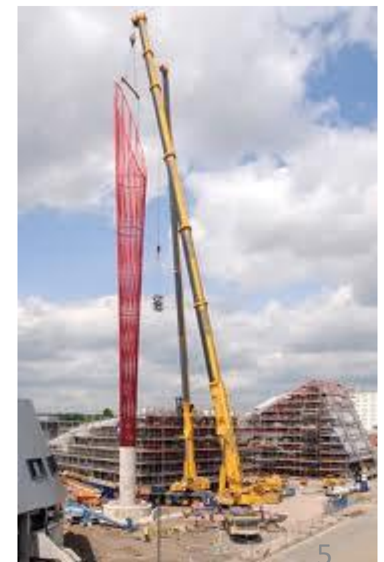
Fit For Purpose

Evolution "designs/generates" organisms for a particular environment.

Similarly we should design metaheuristics for particular problem class.

"we propose a new crossover operator…"

"what is it for…"



Problem Instances and Classes

- A **problem instance** is an instance of a given type of problem e.g. a real valued function over bit strings of length n.
  - E.g. Hamming-Distance(x, t) the hamming distance between x and a fixed target t.
- A **problem class** is a probability distribution over problem instances.
  - E.g. the bit strings t are drawn from a fixed probability distribution.
- We can learn at the **instance** level or at the **class** level.
  - Most systems concentrate on learning at the instance level.
  - In this paper we learn at the class level (i.e. exploitable properties of the class).
- We will **design heuristics for a problem class**.

No Free Lunch Theorems

- NFL theorems says *no metaheuristic performs better over all problem instances when compared to another metaheuristic*.
- It also implies that a metaheuristic that does better on one class of problems must do worse on another class of problems.
- Therefore we **MUST** design our metaheuristics for a specific problem class.

Bias and Meta bias

- Bias of a metaheuristic is basically a probability distribution over a search space.
- For a given metaheuristic this is static.
- If the bias of a metaheuristic does not match our problem class we have no mechanism to change it.
- Meta bias provides a method to alter bias.
- **Meta bias is necessary if we are to apply our algorithm to multiple instances of a problem.**
- This is what the proposed method does.

Why automatically design metaheuristics?

- **Faster design** than human design (free**r** of implicit and unconscious design decisions made by humans)
- **Better performance** than human designed heuristics (guaranteed)
- **Tailored to a specific problem class** (we make no guarantees on performance on other problem classes).

Generic Algorithms

- Standard metaheuristics need to be executed each time on each problem instance and produce a solution to that instance.
- A generic algorithm is a general solution to a problem class.
- ***Generic Algorithm =***

  Genetic Programming + Application Framework

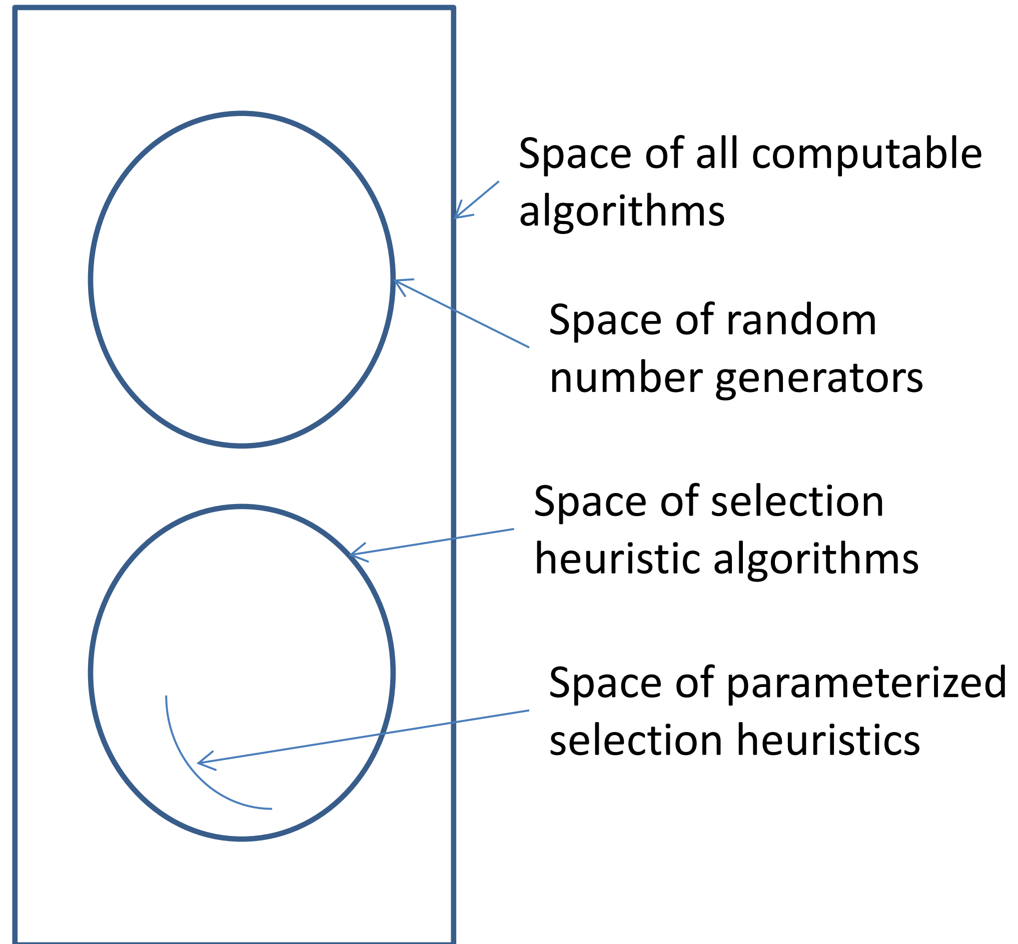
  Genetic Programming provides the "algorithms"

  The application framework provides the platform in which the algorithms are executed and applied to the problem.

  Applied to TSP, SAT, bin-packing

Program Space

- A program space defines the search space to which we are confined.
- The space of "all algorithms" is too large – it includes e.g. random number generators.
- The space of parameterized algorithms is too small – it only includes a linear weighted sum.
- We can restrict our search to algorithms of interest.



Human Designed Selection Heuristics

- **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.

Current population (index, fitness, bit-string)

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

0001010 0111010 0001010 0100010

Next generation

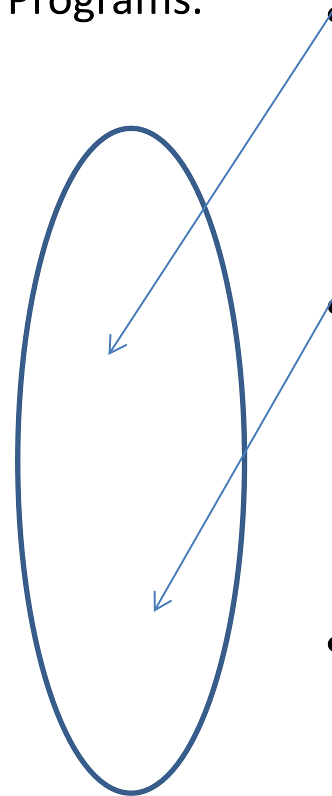
Framework for Selection Heuristics

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
- Register Machines calculate `value( p )` and are used to generate a new population from old.

Space of URM Programs.



- rank selection is the program. `Copy R1 R0`
- fitness proportional selection is the program `Nop`
- These are just two programs in our search space.

Register Machines

- A program is a list of instructions
- A program acts on a register.
- Inputs and outputs are communicated to the program via the register.
- R0=fitness
- R1=index i.

Program

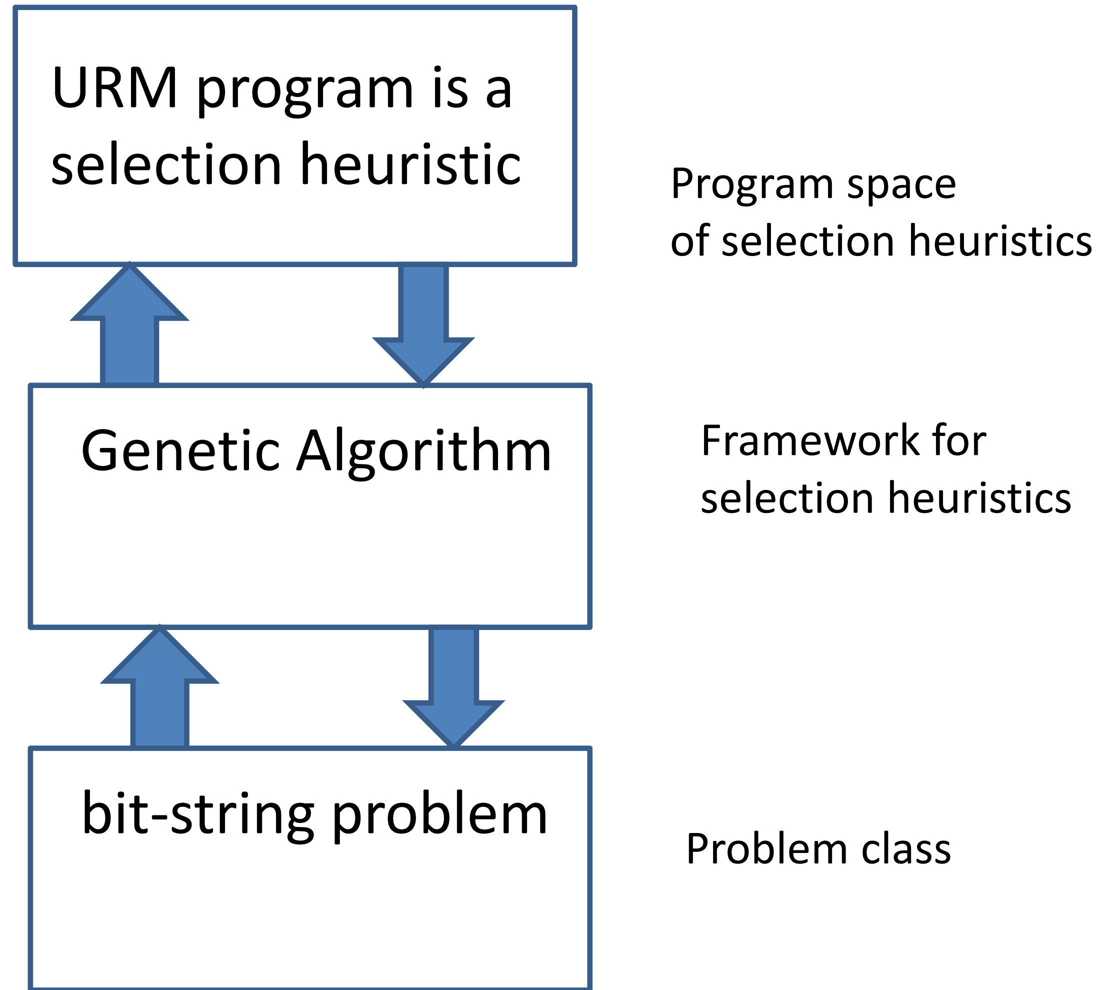
PC	Instruction
1	Inc R1
2	Copy R1 R0
3	Set R1 0.33

Registers after each instruction

PC	R0	R1	R2
0 (initial)	2.9	3	5
1	2.9	4	5
2	4	4	5
3 (final)	4	0.33	5

URM evaluation

- URMs are generated by random search in the top layer.
- URMs are passed to the lower level where they are used as a selection heuristic on in a GA on a bit string problem class.
- A value is passed to the upper layer informing it of how well the URM performed as a selection heuristic.



Experiments

- Train on 50 problem instances (i.e. we run a single Register Machine for 50 runs of a genetic algorithm on a mimicry problem instance from our problem class).
- The training times are ignored
  - we **are not comparing** our search method of register machines.
  - We **are comparing** our selection heuristic with rank and fitness proportional selection.
- Selection heuristics are tested on a second set of problem instances drawn from the same problem class.

Parameter settings for GA

Parameter	Value
num-bits	64
metaheuristic-num-runs	50
metaheuristic-population-size	30
metaheuristic-num-generations	50
metaheuristic-mutation-probability	0.1

Parameter Values for Register Machine Search

Parameter	Value
Random-search-iterations	100
RM program length	2
register size	3
output register	*R0*
contents of *R0*	*fitness*
contents of *R1*	*rank*
contents of *R2*	*0 (working register)*

Instruction Set

Instruction	Action	Arguments
Inc	$Ri \leftarrow Ri + 1$	1
Dec	$Ri \leftarrow Ri - 1$	1
Add	$Rk \leftarrow Ri + Rj$	3
Sub	$Rk \leftarrow Ri - Rj$	3
Mul	$Rk \leftarrow Ri * Rj$	3
Div	$Rk \leftarrow Ri = Rj \text{ if } Rj \neq 0; 0$	3
Set	$Ri \leftarrow x \in R$	2
Copy	$Ri \leftarrow Rj$	2
Clear	$Ri \leftarrow 0$	1
Swap	$Ri \leftrightarrow Rj$	2

Problem Classes

1. Generate values N(0,1) in interval [-1,1] (if we fall outside this range we regenerate)
2. Interpolate values in range [0, 2^{num-bits}-1]
3. 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.

Results

	Fit Prop	Rank	RM-select
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 RM-selection resulted in a p-value of better than 10^{*15} in both cases. In both of these* cases the RM outperforms the standard selection operators.

Take Home Points

- Contribution is a mechanism for automatically designing selection heuristics.
- We should design metaheuristics for classes of problems i.e. with a context/niche.
- This approach is human competitive and human cooperative.
- Meta bias is necessary if we are to tackle multiple problem instances.
- Think frameworks not algorithms – we don’t want to solve problem instances we want to solve classes!