4th ECADA
Evolutionary Computation for the Automated Design of Algorithms
GECCO WORKSHOP 2014
John Woodward jrw@cs.stir.ac.uk
Jerry Swan jsw@cs.stir.ac.uk
Earl Barr E.Barr@ucl.ac.uk
Welcome + Outline

- Schedule
- Proposing many algorithm
- Template method + type signatures
- Base and meta-level learning
- Problem Classes (probability of an instance)
- Example: Bin Packing
- Outlook.
Schedule

- **8:30-9:00** Introduction and Overview.
- **9:00-9:30** Automated Design of Algorithms and Genetic Improvement: Contrast and Commonalities
  - Saemundur O. Haraldsson, John R. Woodward
- **9:30-10:00** Benchmarks that Matter for Genetic Programming
  - John R. Woodward, Simon P. Martin, Jerry Swan
- **10:00-10:40** Coffee Break
- **10:40-11:10** A Problem Configuration Study of the Robustness of a Black-Box Search Algorithm Hyper-Heuristic
  - Matthew A. Martin, Daniel R. Tauritz
- **11:10-11:40** A Step Size Based Self-Adaptive Mutation Operator for Evolutionary Programming
  - Libin Hong, John H. Drake, Ender Ozcan
- **11:40-12:10** Discussion
Schedule < coffee

• **8:30-9:00** Introduction and Overview.
• **9:00-9:30** Automated Design of Algorithms and Genetic Improvement: Contrast and Commonalities
  • Saemundur O. Haraldsson, John R. Woodward
• **9:30-10:00** Benchmarks that Matter for Genetic Programming
  • John R. Woodward, Simon P. Martin, Jerry Swan
• **10:00-10:40** Coffee Break
Schedule > coffee

- **10:00-10:40** Coffee Break
- **10:40-11:10** A Problem Configuration Study of the Robustness of a Black-Box Search Algorithm Hyper-Heuristic
  - Matthew A. Martin, Daniel R. Tauritz
- **11:10-11:40** A Step Size Based Self-Adaptive Mutation Operator for Evolutionary Programming
  - Libin Hong, John H. Drake, Ender Ozcan
- **11:40-12:10** Discussion
Conceptual Overview

Combinatorial problem e.g. TSP salesman
Exhaustive search?

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.

10/07/2014 John Woodward University of Stirling
One Man – One 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...
3. What is the target problem class they have in mind?

Human 1 → Heuristic Algorithm 1
Human 2 → Heuristic Algorithm 2
Human 3 → Heuristic Algorithm 3
One Man .... Many Algorithms

1. Challenge is defining an algorithmic framework (set) that includes useful algorithms, and excludes others.

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.
Proposing Sets of Algorithms

In the *template method*, one or more algorithm steps can be **overridden** by subclasses to allow **differing** behaviours while ensuring that the **overarching** algorithm is still followed.

• **Concrete** methods/classes **constrain** the **behaviour** of the program.

• **Abstract** methods/classes allow variation.

A template is a **skeleton**.
Template Method Hyper-heuristics

• **Template Method** is a design pattern.

• Some methods of a class have specified type signatures but no implementation (body) i.e. **abstract class**.

• The abstract class(es) can be **supplied later** by another programmer.

• OR can be supplied by Automatic Programming Technique such as **Genetic Programming**.
Type Signatures

- $H = \text{history}$, $P = \text{population}$

\[
\begin{align*}
\text{initialization} & : \text{Void} \rightarrow P \\
\text{selection} & : P \times H \rightarrow P \\
\text{variation} & : P \times H \rightarrow P \\
\text{succession} & : P \times H \rightarrow P \\
\text{termination} & : H \rightarrow \text{Bool}
\end{align*}
\]
procedure evolve
begin
    pop = initialization()
    history = []
repeat
    parents = selection( pop, history )
    offspring = variation( parents, history )
    pop = succession( offspring, history )
    history = history.append( pop )
until termination( history )
end
Example – mutation for GA.

• **Examples:** one point and uniform mutation.
• **Behaviour:** Given a bit string of length $n$, return a bit string of length $n$.
• We could write another mutation operator.
• **NO NO NO** – lets let Genetic Programing DO ALL THE HARD (and boring) WORK.
• **Generate-and-test a Generate-and-test method**
Meta and Base Learning

1. At the **base** level we are learning about a **specific** function.
2. At the **meta** level we are learning about the problem **class**.
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

14 John Woodward University of Stirling
<table>
<thead>
<tr>
<th>Genetic Algorithm</th>
<th>Genetic Algorithm Designer</th>
</tr>
</thead>
<tbody>
<tr>
<td>• (B^n -&gt; R) -&gt; B^n</td>
<td>• [(B^n -&gt; R)] -&gt; ((B^n -&gt; R) -&gt; B^n)</td>
</tr>
</tbody>
</table>

**Input** is an objective function mapping bit-strings of length n to a real-value.  
**Output** is a (near optimal) bit-string i.e. the solution to the problem instance  

**Input** is a *list of* functions mapping bit-strings of length n to a real-value (i.e. sample problem instances from the problem class).  
**Output** is a (near optimal) mutation operator for a GA i.e. the solution method (algorithm) to the problem class

We are raising the level of generality at which we operate.  
Give a man a fish and he will eat for a day, teach a man to fish and...
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.
In a Nutshell

• **Humans** design the **structure** of the program e.g. the for-loops (GP is bad at that) (INVARIANT)
• Let **GP** build the **body** of the for-loop (VARIANT).
• The final program is part **man made** and part **machine made**.
• We used the **Object Oriented** approach but could be expressed in terms of e.g. **Functional programming** (pass in a mutation operator).
Problem Classes Do Occur

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

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

3. Problem classes do occur in the real-world

4. Next 6 slides demonstrate problem classes and scalability with on-line bin packing.
On-line Bin Packing

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

Array of bins

Pieces packed so far

150 = Bin capacity

Range of piece size 20-100

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

Not immediately obvious how to link Genetic Programming to apply to combinatorial problems. See previous paper. The GP tree is applied to each bin with the current piece put in the bin which gets maximum score.

Terminals supplied to Genetic Programming

- Initial representation \{C, F, S\}
- Replaced with \{E, S\}, E = C - F
- We can possibly reduce this to one variable!!
How the heuristics are applied
Robustness of Heuristics

= all legal results
= some illegal results
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.
Compared with Best Fit

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

Amount the heuristics beat best fit by:

Number of pieces packed so far:

-100 0 100 200 300 400 500 600 700

0 20000 40000 60000 80000 100000

Amount evolved heuristics beat best fit by.

evolved on 100
evolved on 250
evolved on 500
Compared with Best Fit

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

1. Image Recognition – Roberts Mark
2. Travelling Salesman Problem – Keller Robert
4. Data Mining – Gisele L. Pappa, Alex A. Freitas
5. Decision Tree - Gisele L. Pappa et. al.
6. Selection Heuristics – Woodward & Swan
7. Bin Packing 1,2,3 dimension (on and off line)
   Edmund Burke et. al. & Riccardo Poli et. al.
A Paradigm Shift?

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

1. Instead of proposing a single algorithm, “In this paper we propose a novel algorithm”...

2. We can now propose a set of algorithms, “In this paper we propose 10,000 algorithms”

3. The resulting algorithm is typically better than a human designed algorithm.

4. If the problem changes, we can instantly call on Genetic Programming again.
Conclusions

1. Algorithms are reusable, “solutions” aren’t (e.g. TSP).
2. We can automatically design algorithms that consistently outperform human designed algorithms (on various domains).
3. 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.
4. We can design algorithms on small problem instances and scale them apply them to large problem instances (TSP, child multiplication).