Hands-on #2: GP for Spectrum Based Fault Localisation

School of Computing, KAIST
Shin Yoo

Outline

- A reproduction(!) of my talk at International Symposium on Search Based Software Engineering (SSBSE) 2012
  - This is both a very successful use of GP for SE and a nice introduction for the second hands-on
- A brief look at GP for symbolic regression using DEAP
- Hands-on: evolving SBFL formulas
- Report of the state of the art in fault localisation (GP still going strong)

Evolving Human Competitive Spectra-Based Fault Localisation Techniques

Shin Yoo, University College London, UK

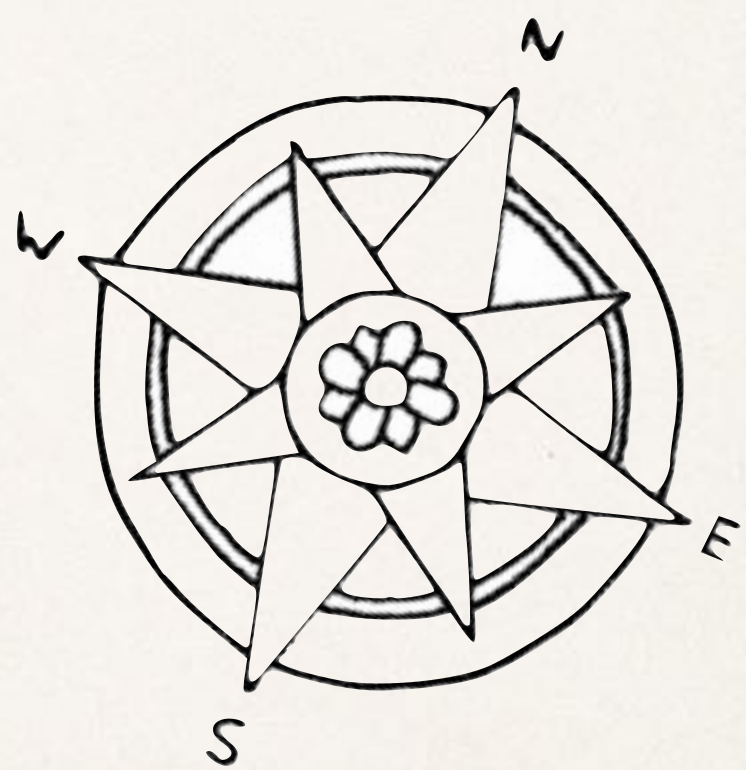
The 4Th International Symposium On Search Based Software Engineering, Riva Del Garda, Italy, 28-30Th September 2012

Overview

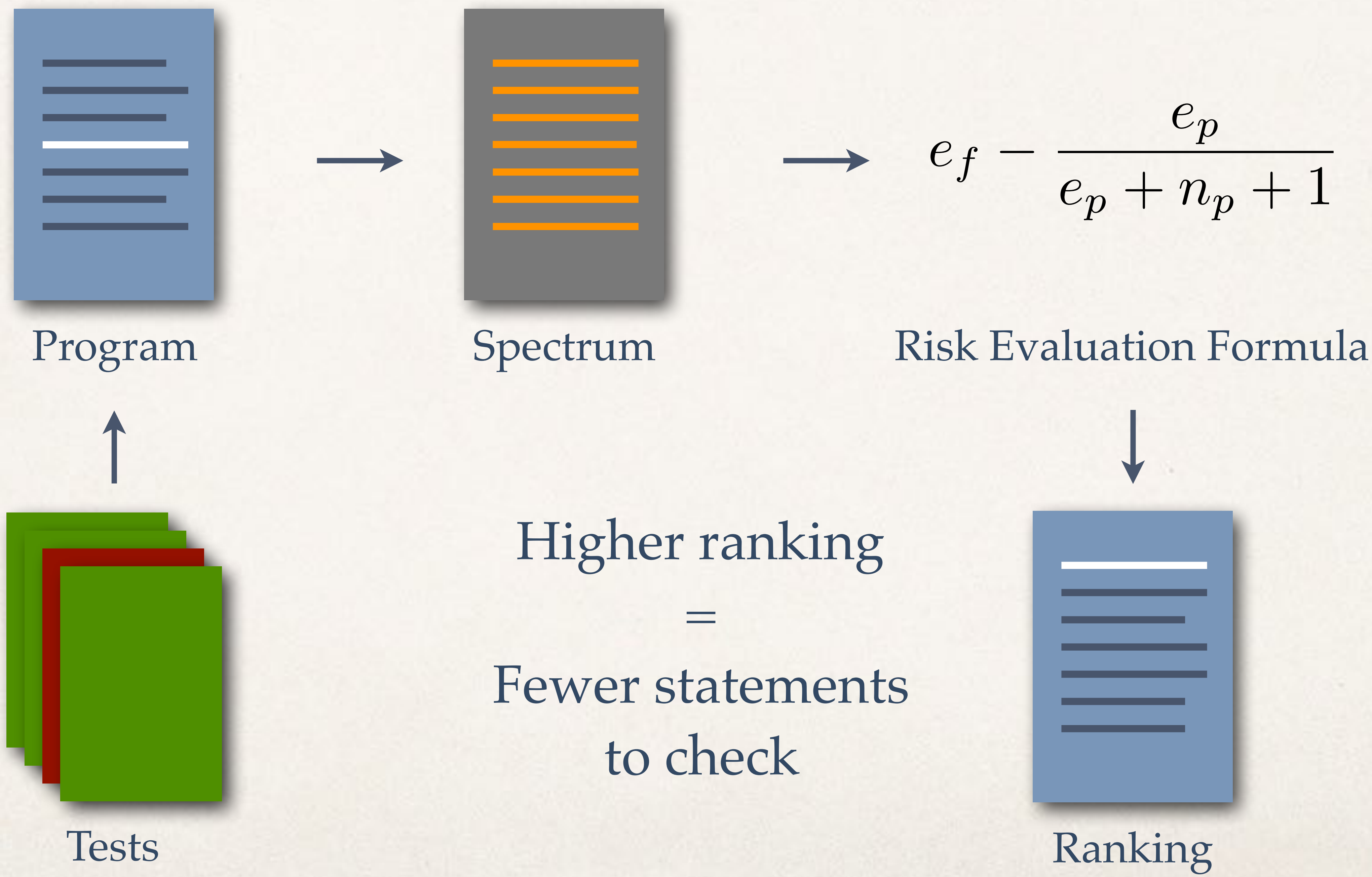
- **Automated debugging** techniques tell you where to look and fix.
- A class of techniques uses **risk evaluation formulæ** to convert program spectra (execution traces) to predicted risk per statement.
- We show that **GP can evolve formulæ** that can outperform humans.

Outline

- Background
  - What is SBFL?
  - The State of the Art
- GP Approach
- Results
- The Way Forward



Spectra Based Fault Localisation



Spectra-Based Fault Localisation

Structural Elements	Test t_1	Test t_2	Test t_3	Spectrum e_p	e_f	n_p	n_f	Tarantula	Rank
s_1	•			1	0	0	2	0.00	9
s_2	•			1	0	0	2	0.00	9
s_3	•			1	0	0	2	0.00	9
s_4	•			1	0	0	2	0.00	9
s_5	•			1	0	0	2	0.00	9
s_6	•		•	1	1	0	1	0.33	4
s_7 (faulty)		•	•	0	2	1	0	1.00	1
s_8	•	•		1	1	0	1	0.33	4
s_9	•	•	•	1	2	0	0	0.50	2
Result	P	F	F						

$$\text{Tarantula} = \frac{\frac{e_f}{e_f+n_f}}{\frac{e_p}{e_p+n_p}+\frac{e_f}{e_f+n_f}}$$

State of the Art

$$\frac{2e_f}{e_f + n_f + e_p}$$

$$\frac{e_f}{e_f + n_p + 2(e_p + n_f)}$$

$$\frac{e_f}{e_f + n_f + e_p}$$

$$\frac{2(e_f + n_p)}{2(e_f + n_p) + e_p + n_f}$$

$$\frac{e_f}{e_f + 2(n_f + e_p)}$$

$$\frac{e_f}{n_f + e_p}$$

Over 30 formulæ in the literature: none guaranteed to perform best for all types of faults

$$\frac{e_f + n_p}{n_f + e_p}$$

$$\frac{e_f}{e_f + n_f + e_p + n_p}$$

$$\frac{e_f + n_p}{e_f + n_f + e_p + n_p}$$

$$\frac{2e_f}{2e_f + n_f + e_p}$$

$$\frac{1}{2}\left(\frac{e_f}{e_f + n_f} + \frac{e_f}{e_f + e_p}\right)$$

$$\frac{\frac{e_f}{e_f+n_f}}{\frac{e_p}{e_p+n_p} + \frac{e_f}{e_f+n_f}}$$

$$\frac{e_f + n_p - n_f - e_p}{e_f + n_f + e_p + n_p}$$

Theoretical Approach I


A Model for Spectra-based Software Diagnosis

LEE NAISH
HUA JIE LEE
and
KOTAGIRI RAMAMOHANARAO
University of Melbourne

1. INTRODUCTION

Optimality Proof (Naish et al. 2011) says

$$Op1 = \begin{cases} -1 & \text{if } n_f > 0 \\ n_p & \text{otherwise} \end{cases} \qquad Op2 = e_f - \frac{e_p}{e_p + n_p + 1}$$

are optimal when the fault lies in:

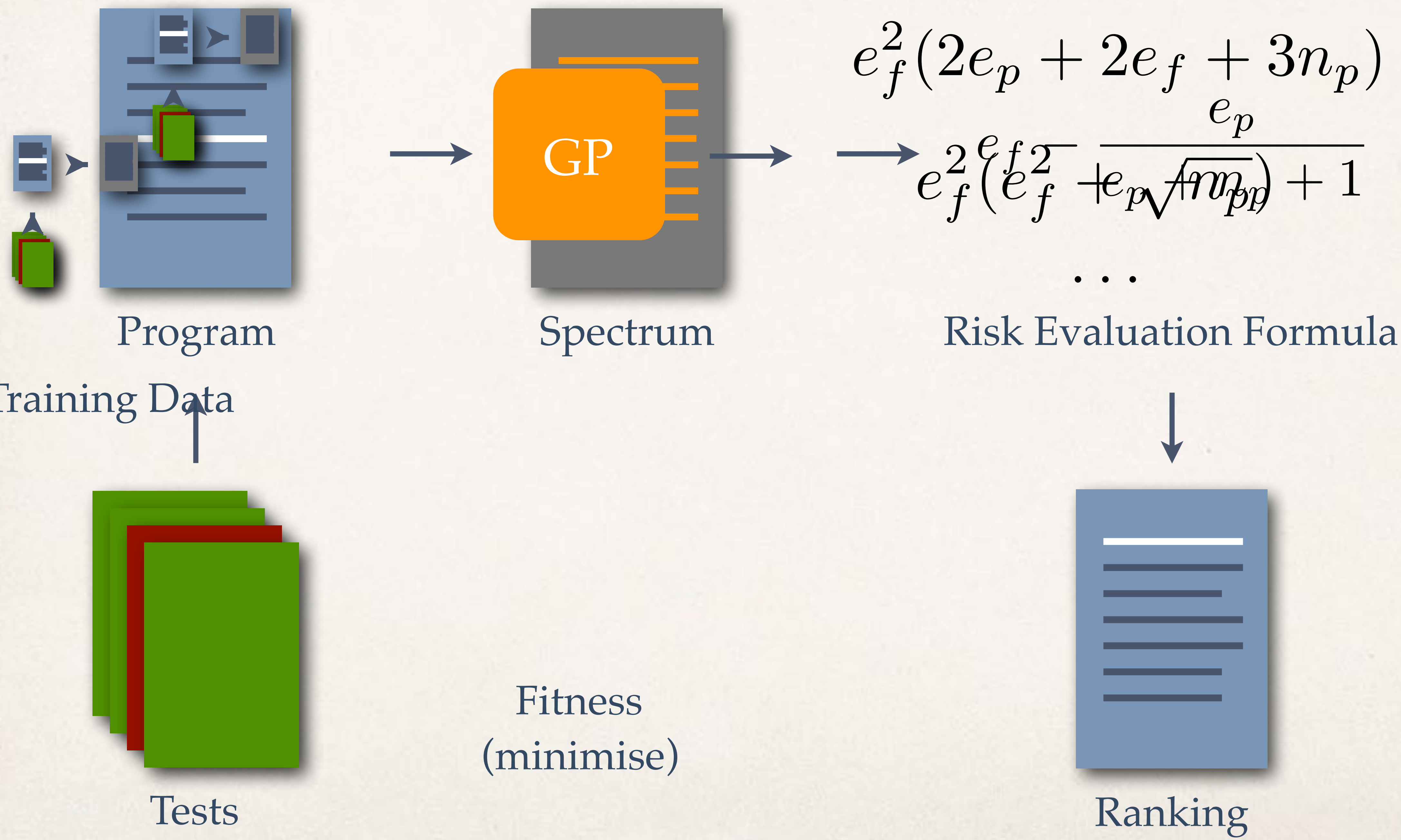
```
if (t1())
    s1();          /* S1 */
else
    s2();          /* S2 */
if (t2())
    x = True;      /* S3 */
else
    x = t3();      /* S4 - BUG */
```

This is hard.

Theoretical Approach II

- Hierarchy & Equivalence: X. Xie 2012 (PhD Thesis)
  - Some techniques are *proven* to outperform others
  - Reduces attack fronts (fewer formulæ to compete against)
  - Still very hard to prove this, if not harder

Evolving Formulæ



The Competition

- We choose 9 formulæ from Naish et al. 2011:
  - Op2 is the known best.
  - Jaccard, Tarantula, Ochiai are widely studied in SE.
  - Wong & AMPLE are recent additions.

$$Op1 = \begin{cases} -1 & \text{if } n_f > 0 \\ n_p & \text{otherwise} \end{cases} \qquad Op2 = e_f - \frac{e_p}{e_p + n_p + 1}$$

$$Jaccard = \frac{e_f}{e_f + n_f + e_p}$$

$$Tarantula = \frac{\frac{e_f}{e_f+n_f}}{\frac{e_p}{e_p+n_p} + \frac{e_f}{e_f+n_f}}$$

$$AMPLE = |\frac{e_f}{e_f + n_f} - \frac{e_p}{e_p + n_p}|$$

$$Ochiai = \frac{e_f}{\sqrt{(e_f + n_f) \cdot (e_f + e_p)}}$$

$$Wong1 = e_f \qquad Wong2 = e_f - e_p$$

$$Wong3 = e_f - h, h = \begin{cases} e_p & \text{if } e_p \leq 2 \\ 2 + 0.1(e_p - 2) & \text{if } 2 < e_p \leq 10 \\ 2.8 + 0.001(e_p - 10) & \text{if } e_p > 10 \end{cases}$$

Experimental Configuration

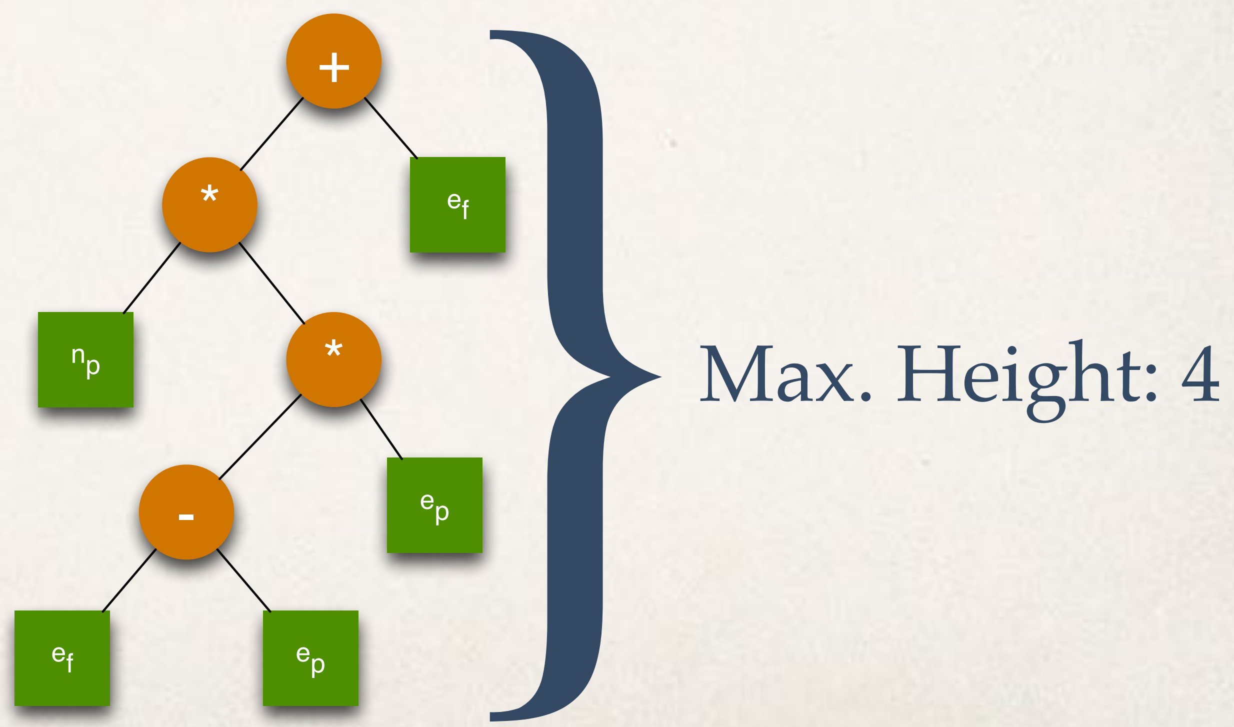
- 92 faults from four Unix tools from SIR
- 30 Runs of Genetic Programming, each with:
  - Random sample of 20 faults as the training set
  - Remaining 72 faults as the evaluation set
  - We sample across programs

Experimental Configuration

- Modified division and square root operator to avoid numerical errors

Operator Node	Definition
`gp_add(a, b)`	`a + b`
`gp_sub(a, b)`	`a - b`
`gp_mul(a, b)`	`ab`
`gp_div(a, b)`	1 if $b = 0$, $\frac{a}{b}$ otherwise
`gp_sqrt(a)`	$\sqrt{\lvert a \rvert}$

- Ramped Initialisation with maximum tree height 4



- Population: 40 / Generations: 100
- Crossover: 1.0 / Mutation: 0.08

Fitness Function/Evaluation

- Expense: normalised ranking of the faulty statement (the lower the better)

$$E(\tau, p, b) = \frac{\text{Ranking of } b \text{ according to } \tau}{\text{Number of statements in } p} * 100$$

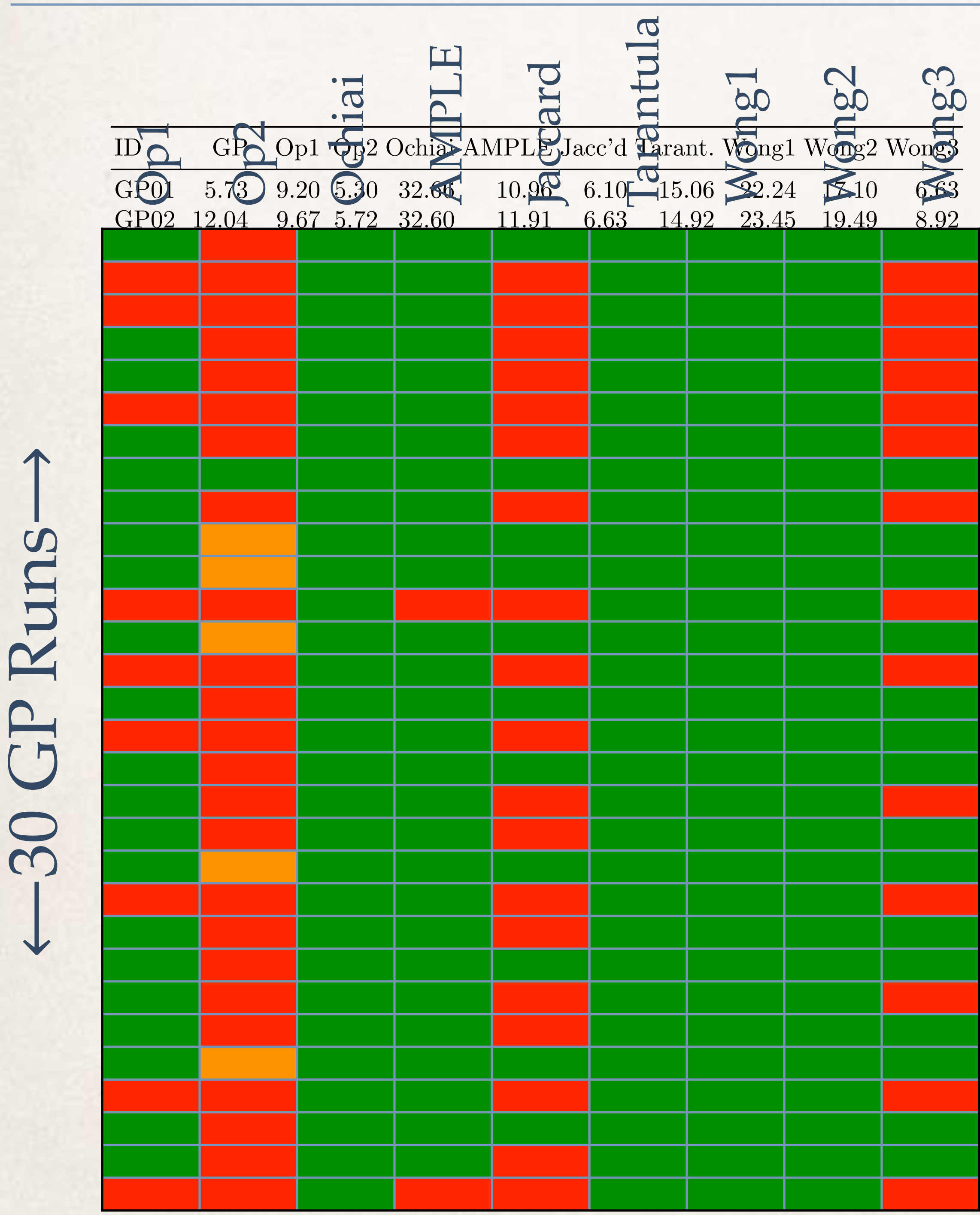
- Fitness: average expense for the 20 faults in the training set

$$\text{fitness}(\tau, B, P) = \frac{1}{n} \sum_{i=1}^{n} E(\tau, p_i, b_i) \text{ (to be minimised)}$$

- Evaluation: average expense for the 72 faults in the evaluation set

Results

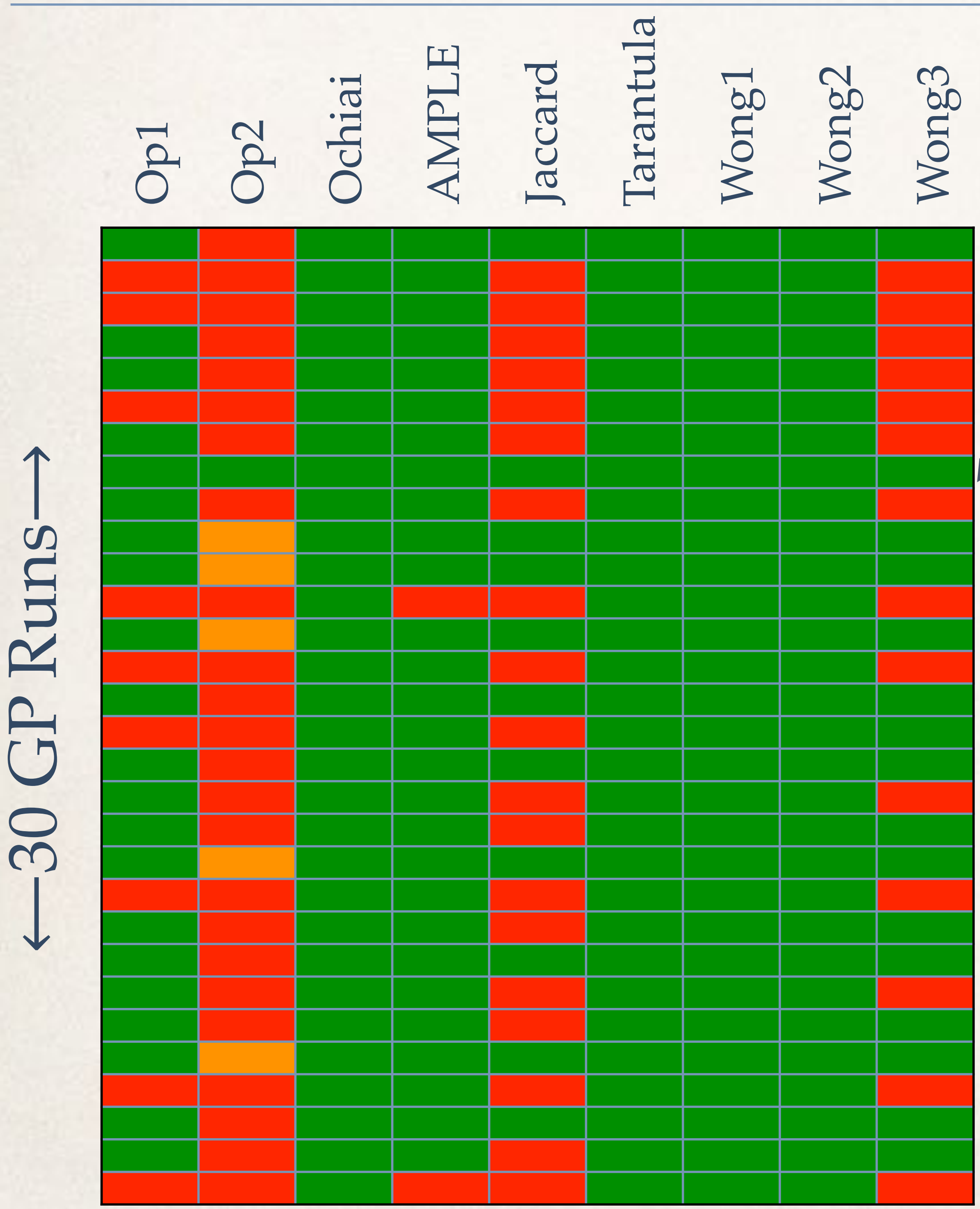
ID	GP	Op1	Op2	Ochiai	AMPLE	Jacc'd	Tarant.	Wong1	Wong2	Wong3
GP01	5.73	9.20	5.30	32.66	10.96	6.10	15.06	22.24	17.10	6.63
GP02	12.04	9.67	5.72	32.60	11.91	6.63	14.92	23.45	19.49	8.92



4 Unix tools w/ 92 faults: 20 for training, 72 for evaluation.

- Green: GP outperforms the other.
- Orange: GP exactly matches the other.
- Red: The other outperforms GP.

Results



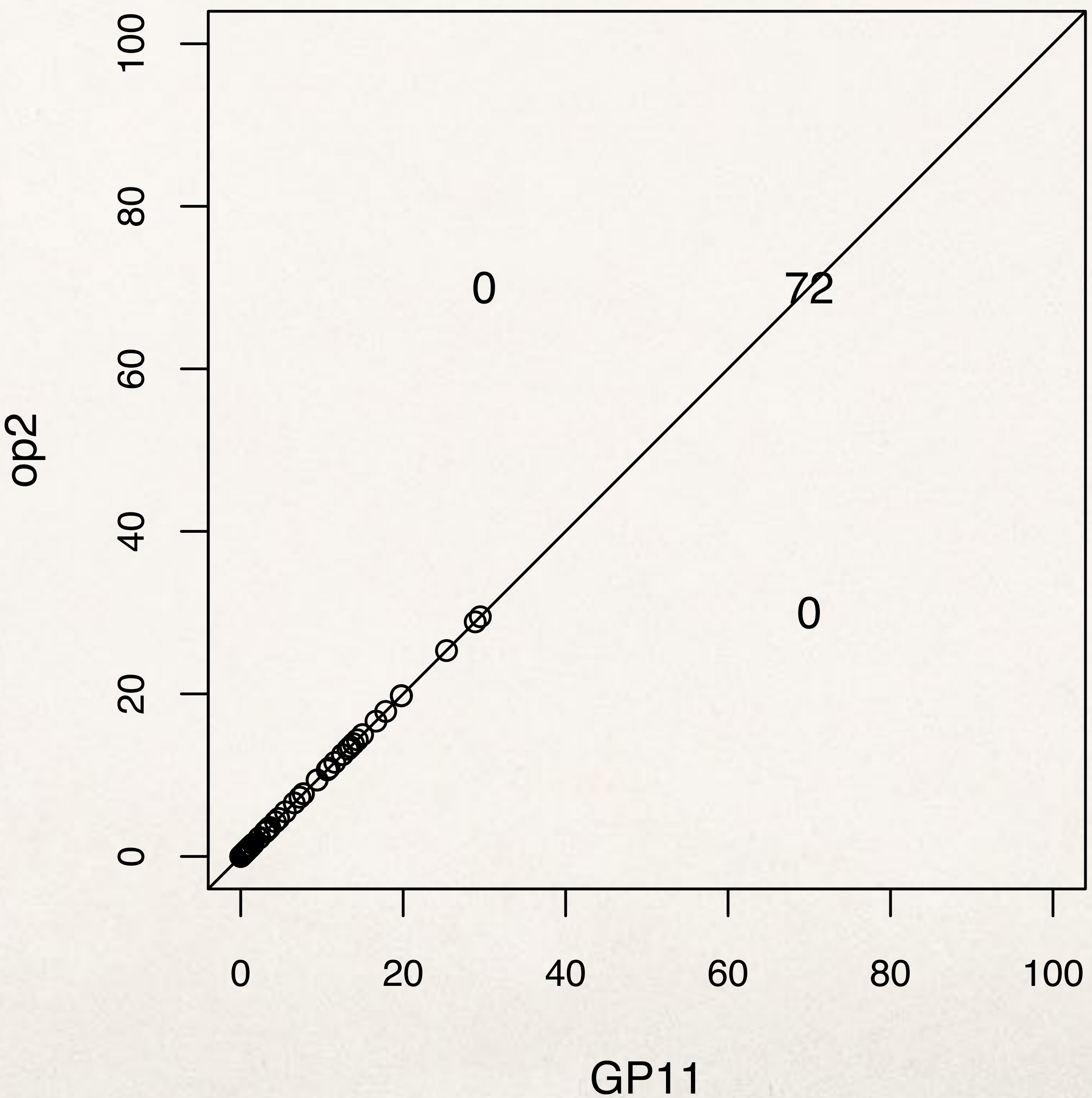
4 Unix tools w/ 92 faults: 20 for training, 72 for evaluation.

My favourite!

$$e_f^2(2e_p + 2e_f + 3n_p)$$

- GP completely outperforms Ochiai, Tarantula, Wong 1 & 2, and mostly outperforms AMPLE.
- Op1, Jaccard, and Wong 3 are tough to beat.
- Op2 is very good but it is not impossible to do better.

Results



Evolved Formulæ

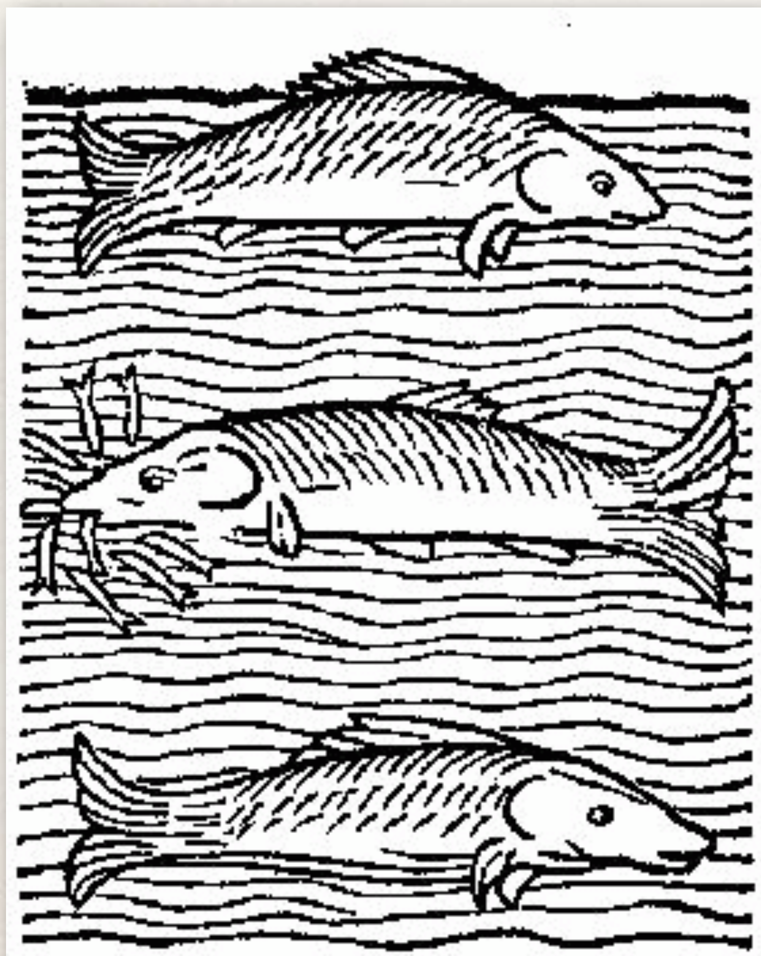
ID	Refined Formula	ID	Refined Formula
GP01	$e_f(n_p + e_f(1 + \sqrt{e_f}))$	GP16	$\sqrt{e_f^{\frac{3}{2}} + n_p}$
GP02	$2(e_f + \sqrt{n_p}) + \sqrt{e_p}$	GP17	$\frac{2e_f + n_f}{e_f - n_p} + \frac{n_p}{\sqrt{e_f}} - e_f - e_f^2$
GP03	$\sqrt{\lvert e_f^2 - \sqrt{e_p}\rvert}$	GP18	$e_f^3 + 2n_p$
GP04	$\sqrt{\lvert \frac{n_p}{e_p - n_p} - e_f\rvert}$	GP19	$e_f\sqrt{\lvert e_p - e_f + n_f - n_p\rvert}$
GP05	$\frac{(e_f + n_p)\sqrt{e_f}}{(e_f + e_p)(n_p n_f + \sqrt{e_p})(e_p + n_p)\sqrt{\lvert e_p - n_p\rvert}}$	**GP20**	$2(e_f + \frac{n_p}{e_p + n_p})$
GP06	$e_f n_p$	GP21	$\sqrt{e_f + \sqrt{e_f + n_p}}$
GP07	$2e_f(1 + e_f + \frac{1}{2n_p}) + (1 + \sqrt{2})\sqrt{n_p}$	GP22	$e_f^2 + e_f + \sqrt{n_p}$
GP08	$e_f^2(2e_p + 2e_f + 3n_p)$	GP23	$\sqrt{e_f}(e_f^2 + \frac{n_p}{e_f} + \sqrt{n_p} + n_f + n_p)$
GP09	$\frac{e_f\sqrt{n_p}}{n_p + n_p} + n_p + e_f + e_f^3$	GP24	$e_f + \sqrt{n_p}$
GP10	$\sqrt{\lvert e_f - \frac{1}{n_p}\rvert}$	GP25	$e_f^2 + \sqrt{n_p} + \frac{\sqrt{e_f}}{\sqrt{\lvert e_p - n_p\rvert}} + \frac{n_p}{(e_f - n_p)}$
GP11	$e_f^2(e_f^2 + \sqrt{n_p})$	**GP26**	$2e_f^2 + \sqrt{n_p}$
GP12	$\sqrt{e_p + e_f + n_p - \sqrt{e_p}}$	GP27	$\frac{n_p\sqrt{(n_p n_f - e_f)}}{e_f + n_p n_f}$
GP13	$e_f(1 + \frac{1}{2e_p + e_f})$	GP28	$e_f(e_f + \sqrt{n_p} + 1)$
GP14	$e_f + \sqrt{n_p}$	GP29	$e_f(2e_f^2 + e_f + n_p) + \frac{(e_f - n_p)\sqrt{n_p e_f}}{e_p - n_p}$
GP15	$e_f + \sqrt{n_f + \sqrt{n_p}}$	GP30	$\sqrt{\lvert e_f - \frac{n_f - n_p}{e_f + n_f}\rvert}$

Insights

- Only 1 formula re-evolved: solution space seems to be large enough.
- Ratio-type components, often found in earlier techniques, do not seem to be essential.
- Similar, intuitively understandable, patterns do emerge:

$$ae_f^x + bn_p^y$$

Way Forward

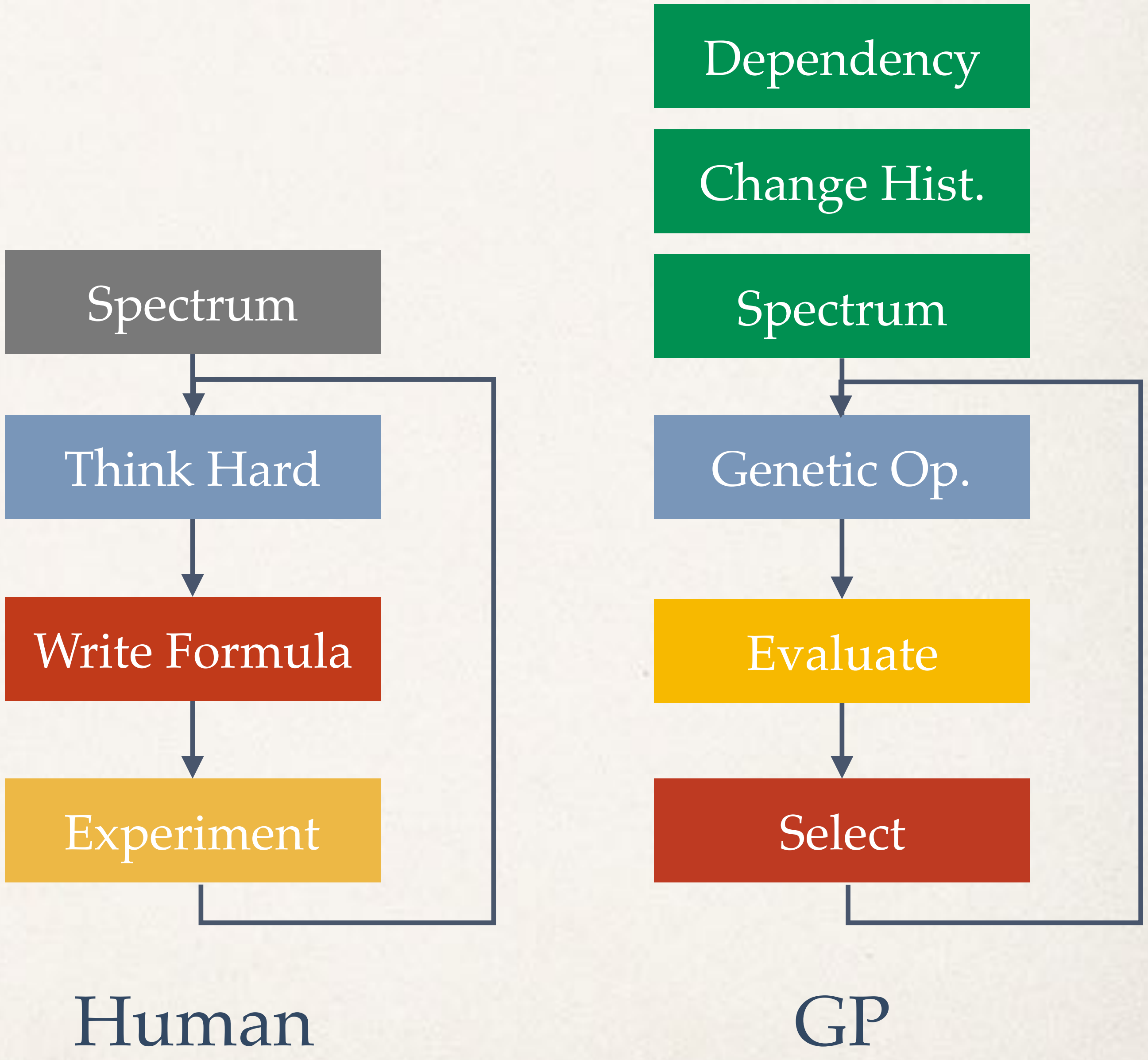


From Solutions to **Generic** Problems...

To Techniques and Strategies for **Your** Problems.

The most effective way to do it, is to do it.

- GP provides a structured, automated way of doing iterative design.
- It can cope with a much diverse spectra and other meta-data.
- GP can evolve a technique that suits **your project**.

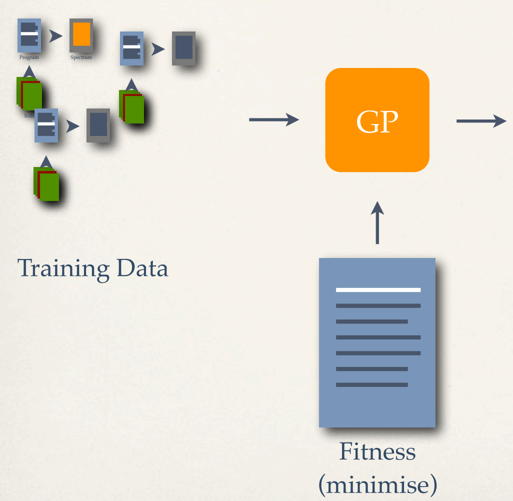


Future Work

- Information Yield: it is not only the ranking that matters! (Yoo et al., TOSEM, to appear)
- Beyond spectrum: metadata from code repository and integration framework
- Parallelisation

http://xkcd.com/917/

Evolving Formulæ

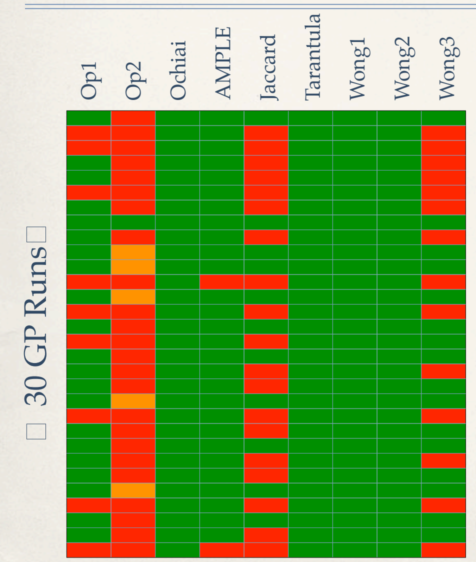


$$e_f^2(2e_p + 2e_f + 3n_p)$$

$$e_f^2(e_f^2 + \sqrt{n_p})$$

$$\ldots$$

Results

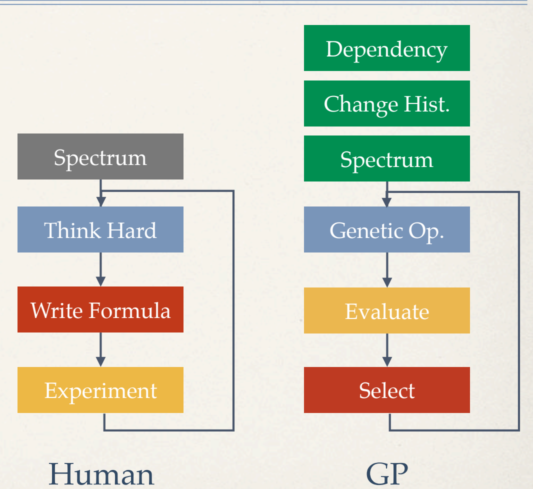


4 Unix tools w/ 92 faults: 20 for training, 72 for evaluation.

- GP completely outperforms Ochiai, Tarantula, Wong 1 & 2, and mostly outperforms AMPLE.
- Op1, Jaccard, and Wong 3 are tough to beat.
- Op2 is very good but it is not impossible to do better.

The most effective way to do it, is to do it.

- GP provides a structured, automated way of doing iterative design.
- It can cope with a much diverse spectra and other meta-data.
- GP can evolve to suit **your project**.



Detailed Statistics & Spectra Data

http://www.cs.ucl.ac.uk/staff/s.yoo/evolving-sbfl.html

Symbolic Regression

- We will use DEAP to write GP based symbolic regression, and use this as the starting point for our second hands-on.
- Recall our walk-through of TSP solver.

Setting node types

```python
# Define new functions
def protectedDiv(left, right):
    try:
        return left / right
    except ZeroDivisionError:
        return 1

pset = gp.PrimitiveSet("MAIN", 1)
pset.addPrimitive(operator.add, 2)
pset.addPrimitive(operator.sub, 2)
pset.addPrimitive(operator.mul, 2)
pset.addPrimitive(protectedDiv, 2)
pset.addPrimitive(operator.neg, 1)
pset.addPrimitive(math.cos, 1)
pset.addPrimitive(math.sin, 1)
pset.addEphemeralConstant("rand101", lambda: random.randint(-1,1))
pset.renameArguments(ARG0='x')
```

Setting up tree-based GA

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", gp.PrimitiveTree, fitness=creator.FitnessMin)

toolbox = base.Toolbox()
toolbox.register("expr", gp.genHalfAndHalf, pset=pset, min_=1, max_=2)
toolbox.register("individual", tools.initIterate, creator.Individual, toolbox.
    expr)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
toolbox.register("compile", gp.compile, pset=pset)
```

Fitness and GP operators

```python
def evalSymbReg(individual, points):
    # Transform the tree expression in a callable function
    func = toolbox.compile(expr=individual)
    # Evaluate the mean squared error between the expression
    # and the real function : x**4 + x**3 + x**2 + x
    sqerrors = ((func(x) - x**4 - x**3 - x**2 - x)**2 for x in points)
    return math.fsum(sqerrors) / len(points),

toolbox.register("evaluate", evalSymbReg, points=[x/10. for x in range(-10,10)]
    )
toolbox.register("select", tools.selTournament, tournsize=3)
toolbox.register("mate", gp.cxOnePoint)
toolbox.register("expr_mut", gp.genFull, min_=0, max_=2)
toolbox.register("mutate", gp.mutUniform, expr=toolbox.expr_mut, pset=pset)
```

Main loop is really the same

```python
def main():
    random.seed(318)

    pop = toolbox.population(n=300)
    hof = tools.HallOfFame(1)

    stats_fit = tools.Statistics(lambda ind: ind.fitness.values)
    stats_size = tools.Statistics(len)
    mstats = tools.MultiStatistics(fitness=stats_fit, size=stats_size)
    mstats.register("avg", numpy.mean)
    mstats.register("std", numpy.std)
    mstats.register("min", numpy.min)
    mstats.register("max", numpy.max)

    pop, log = algorithms.eaSimple(pop, toolbox, 0.5, 0.1, 40, stats=mstats,
                                   halloffame=hof, verbose=True)
    # print log
    return pop, log, hof

if __name__ == "__main__":
    main()
```