HERIOT WATT UNIVERSITY

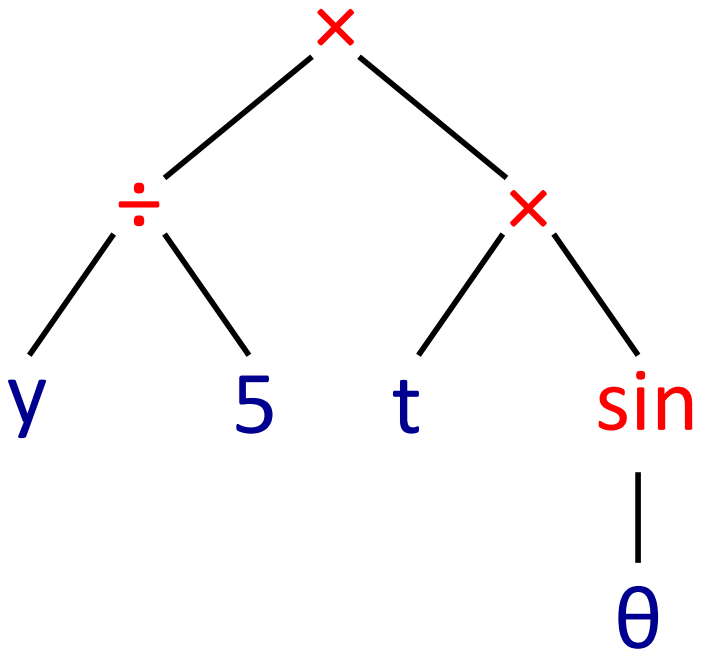
Genetic Programming II

Dr. Michael Lones
Room EM.G31
M.Lones@hw.ac.uk

Previous Lecture

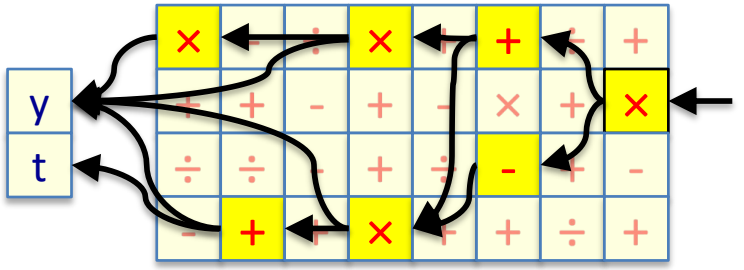
- What GP is and when you should use it
  - Evolutionary algorithms that optimise programs
  - Useful for discovering novel solutions to problems
  - Or solutions to problems you don't know how to solve

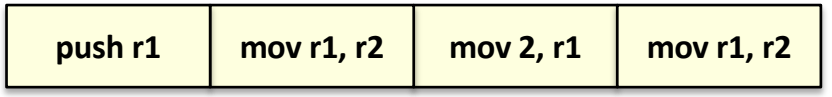
- Introduction to tree-based GP:
  - This is the best known form of GP
  - Some example applications
  - Some issues



Today's Lecture(s)

- An overview of more recent advances in GP
- Some other kinds of GP
  - Cartesian GP
  - Linear GP
  - Grammatical evolution





- Genetic improvement
  - Applying GP to *existing* programs
  - A current hot topic in GP research

Evolvability

This is the capacity for a program to improve its fitness as a result of an evolutionary process (i.e. mutation and recombination).

For genetic programming, there's little value in being theoretically able to express a program if it can not be discovered by evolution.

Expressiveness

This is the capacity for a program representation to express different kinds of behaviours.

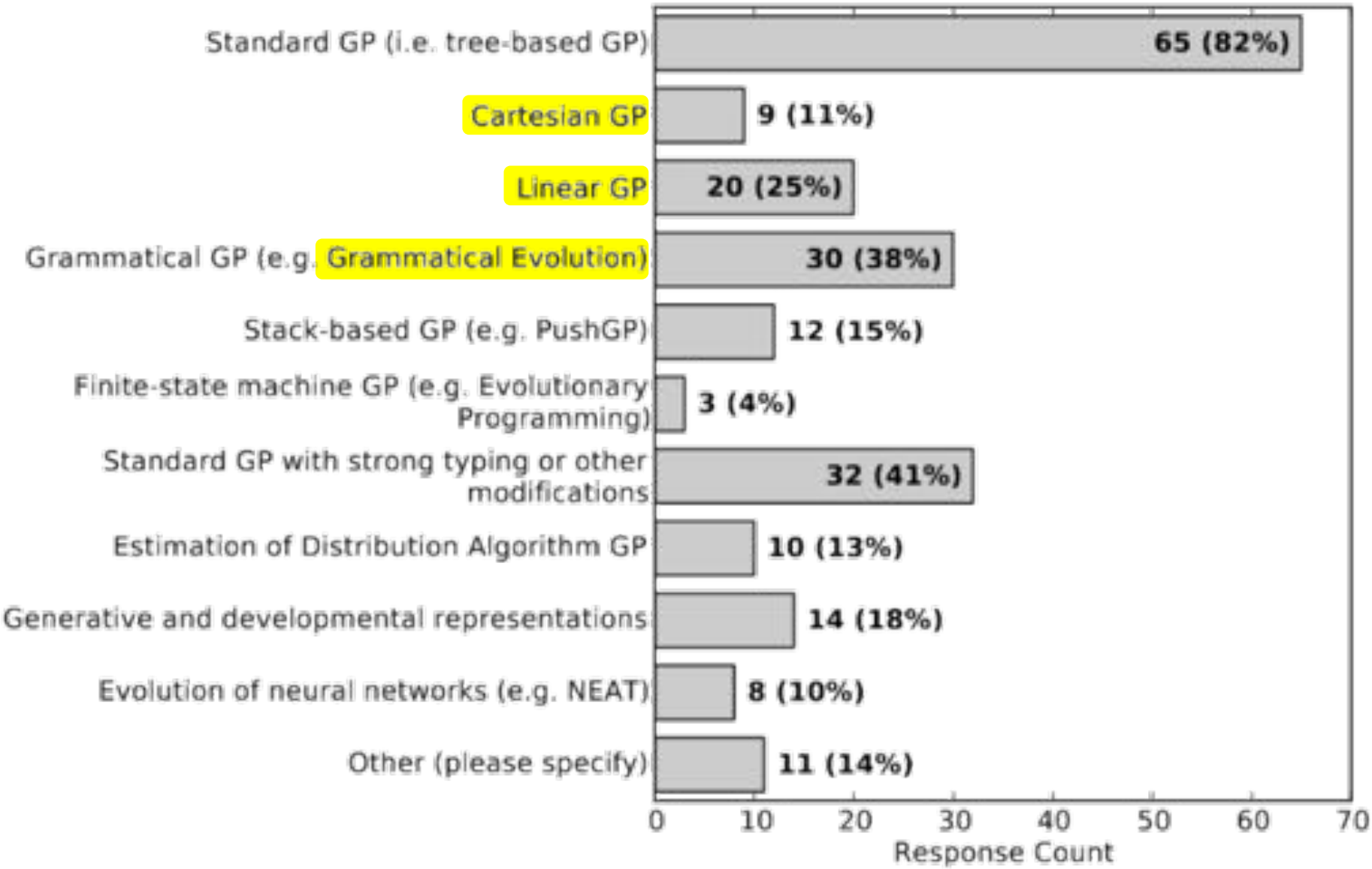
For genetic programming, you can’t evolve a program if you can’t express it.

In practice, there is often a trade-off between expressiveness and evolvability.

Limitations of Koza GP

- Koza tree-based GP is the best known form of GP, but it has some important limitations
  - Adding syntax makes programs more fragile, so it's hard to integrate common things like if statements and loops
  - It doesn't generate programs in a real programming language, making its code difficult to integrate
  - It tends to generate overly-complex expressions, in part because it doesn't allow sub-expression reuse
  - There have been partial solutions to these problems, e.g. ADFs [Field Guide] but they tend to be clunky and fragile

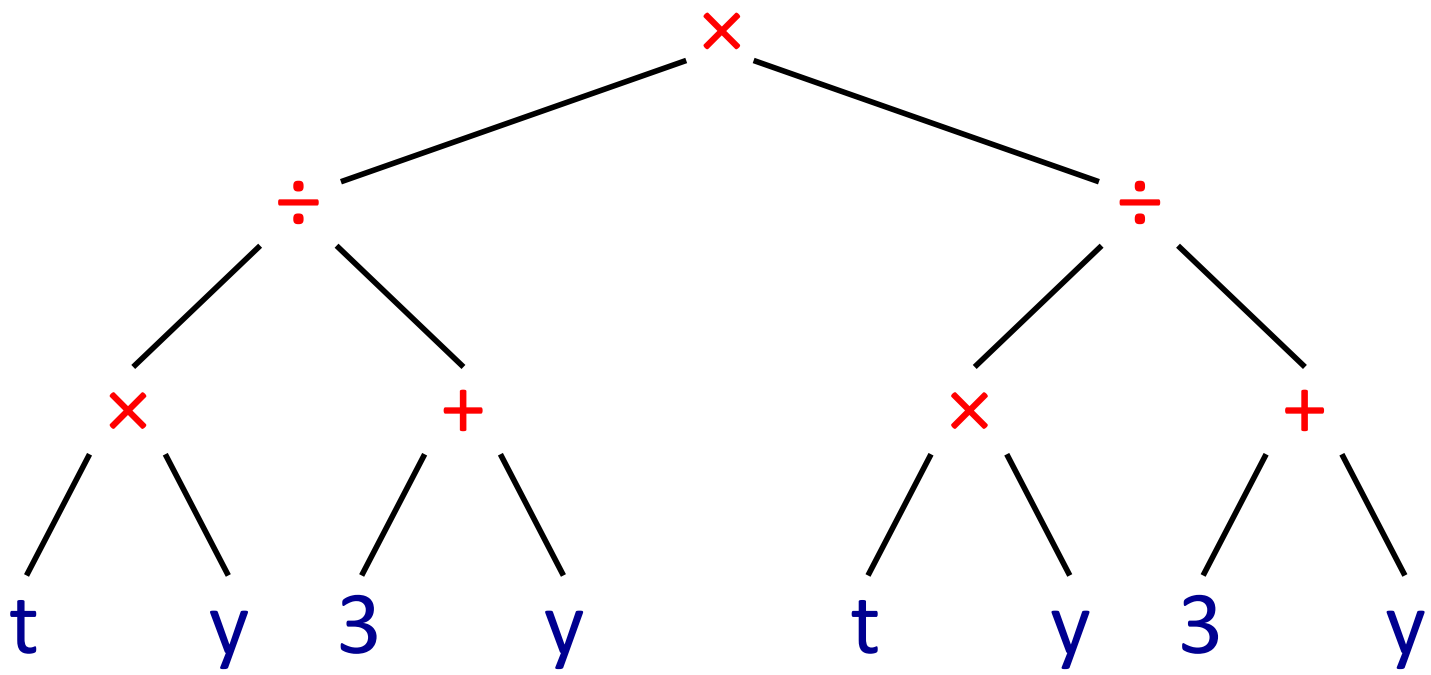
GP Community Survey (2013)



http://link.springer.com/article/10.1007%2Fs10710-012-9177-2

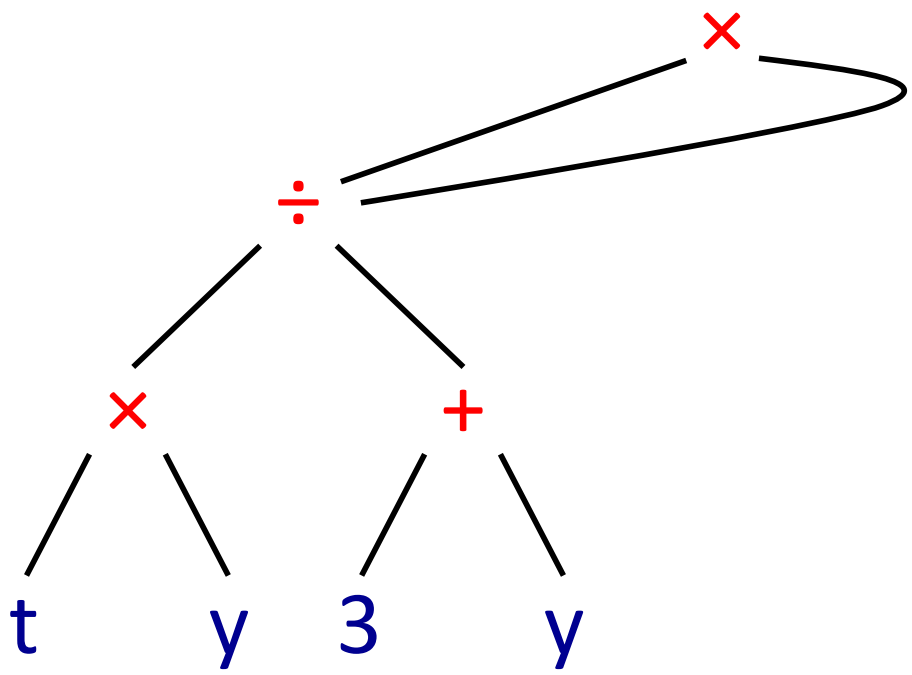
Going Graphical

- Some limitations of GP are due to using trees
  - Imagine this tree is the best solution to some problem;
  - if so, the same sub-tree must be evolved twice:



Going Graphical

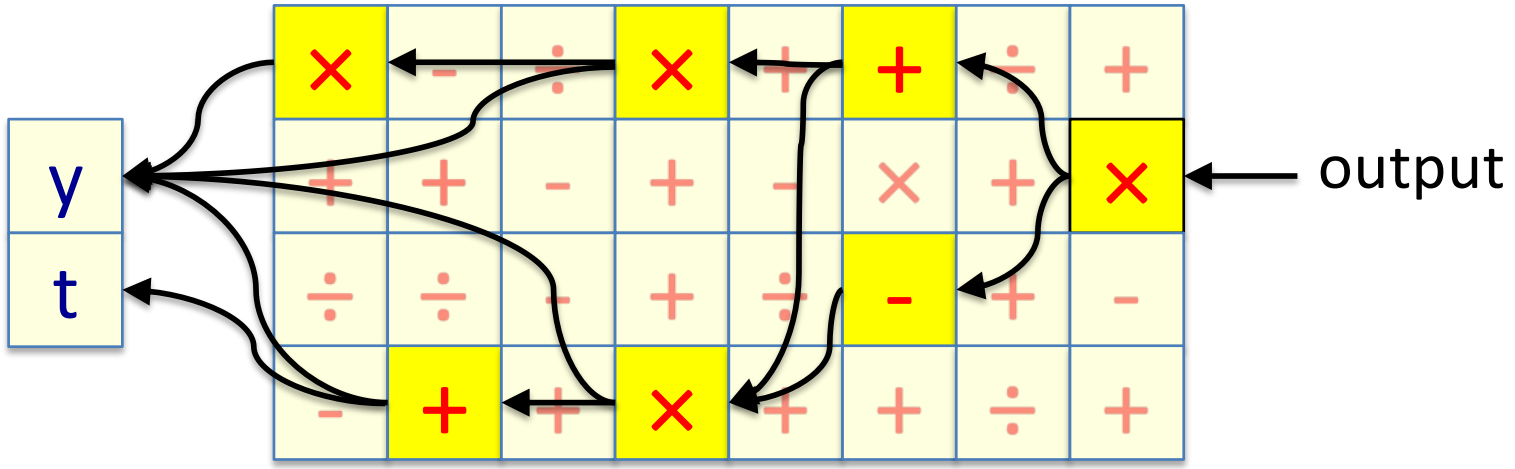
- There are advantages to using graphs instead
  - Unlike trees, graphs can have cycles
  - Instant reuse!



Going Graphical

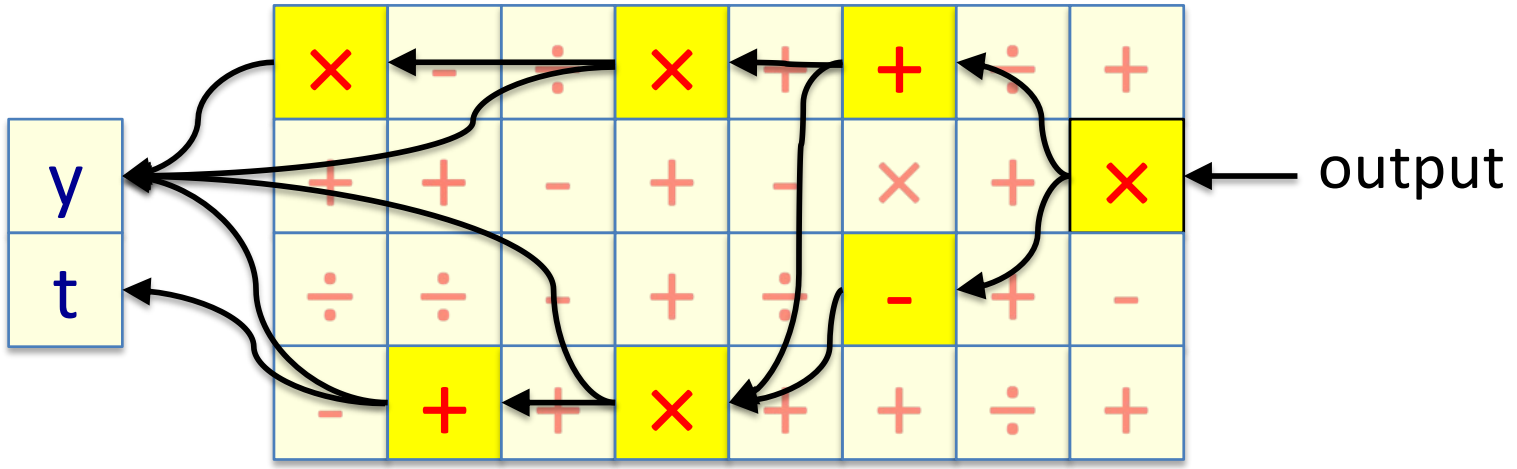
- There are a number of graph-based GPs
  - PADO, PDGP, GNP, CGP, …

- Cartesian GP (CGP) is the best known [Miller 2000]
  - Functions are arranged on a grid (a “Cartesian plane”)



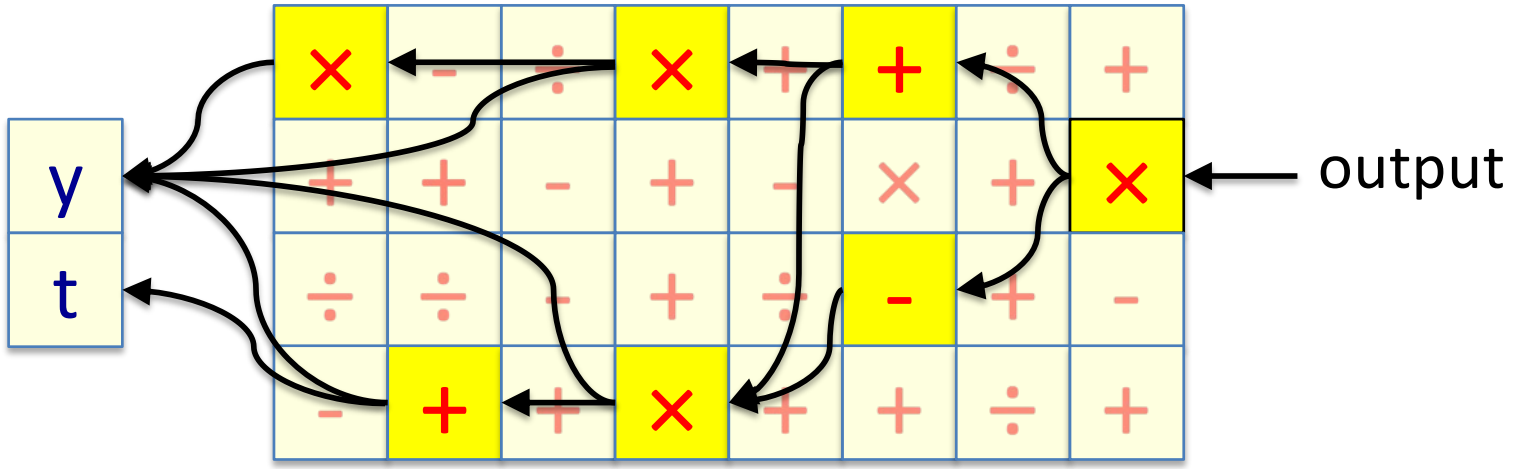
Cartesian GP

- Other than being graph-based, a notable property of CGP is the use of a fixed-size grid
  - Which restricts program size, meaning that you don't have to worry about bloat
  - Though you do need to choose a grid that's big enough, but not too big (since this increases the search space)



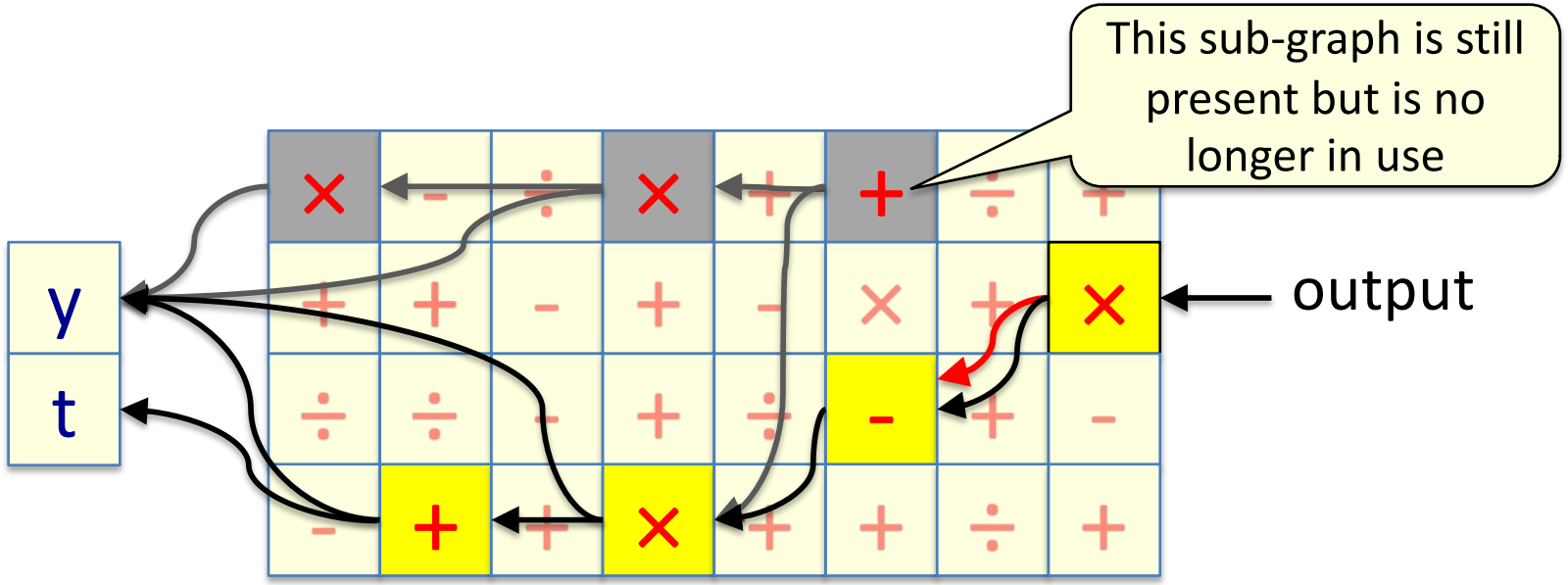
Cartesian GP

- Another important property is that many of the functions in the grid are unused during execution
  - Because they're not connected to the output(s)
  - However, mutation changes connections, meaning that these nodes go in and out of use over evolutionary time



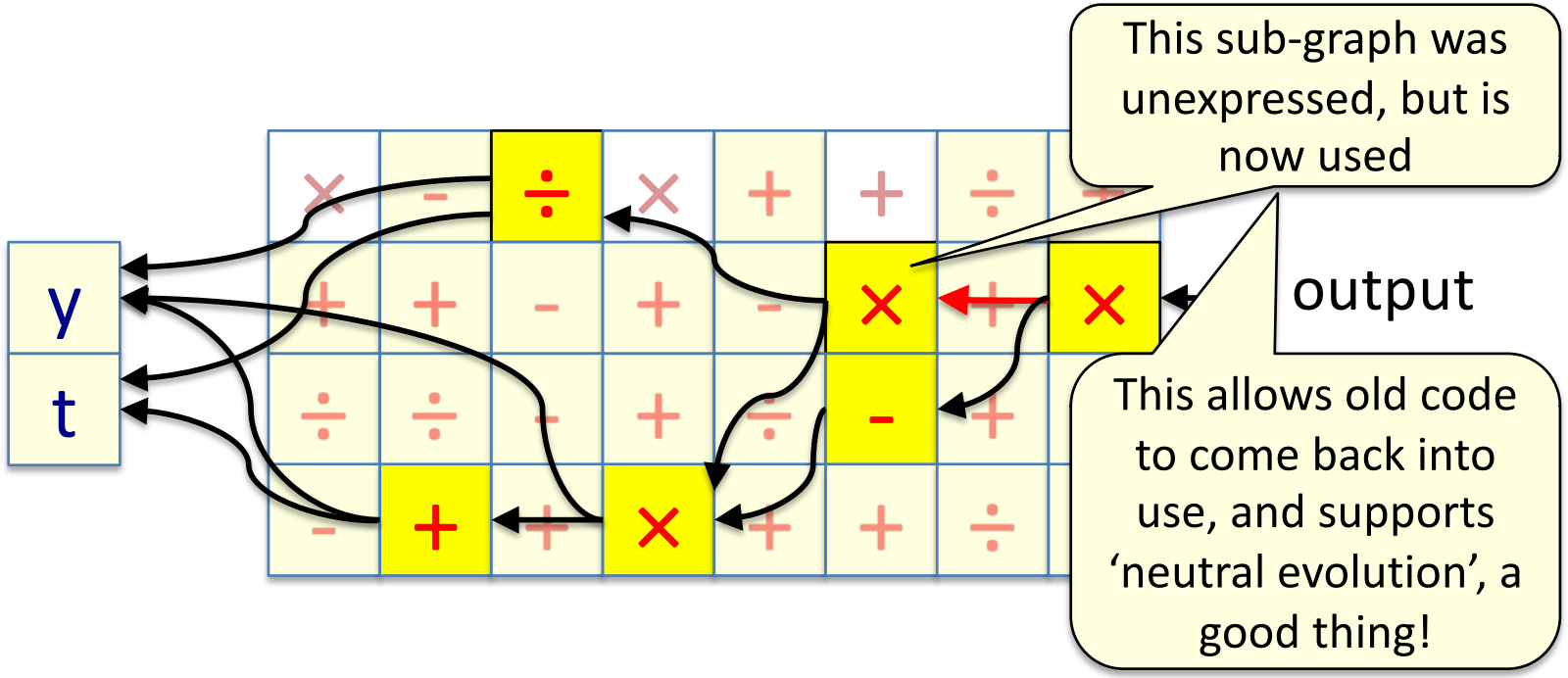
Cartesian GP

- Another important property is that many of the functions in the grid are unused during execution
  - Because they're not connected to the output(s)
  - However, mutation changes connections, meaning that these nodes go in and out of use over evolutionary time



Cartesian GP

- Another important property is that many of the functions in the grid are unused during execution
  - It has been suggested that this allows CGP programs to evolve more like biological systems, allowing old genetic material to be re-used in new contexts



CGP vs Deep Learning

HERIOT WATT UNIVERSITY

https://www.technologyreview.com/s/611568/evolutionary-algorithm-outperforms-deep-learning-machines-at-video-games/

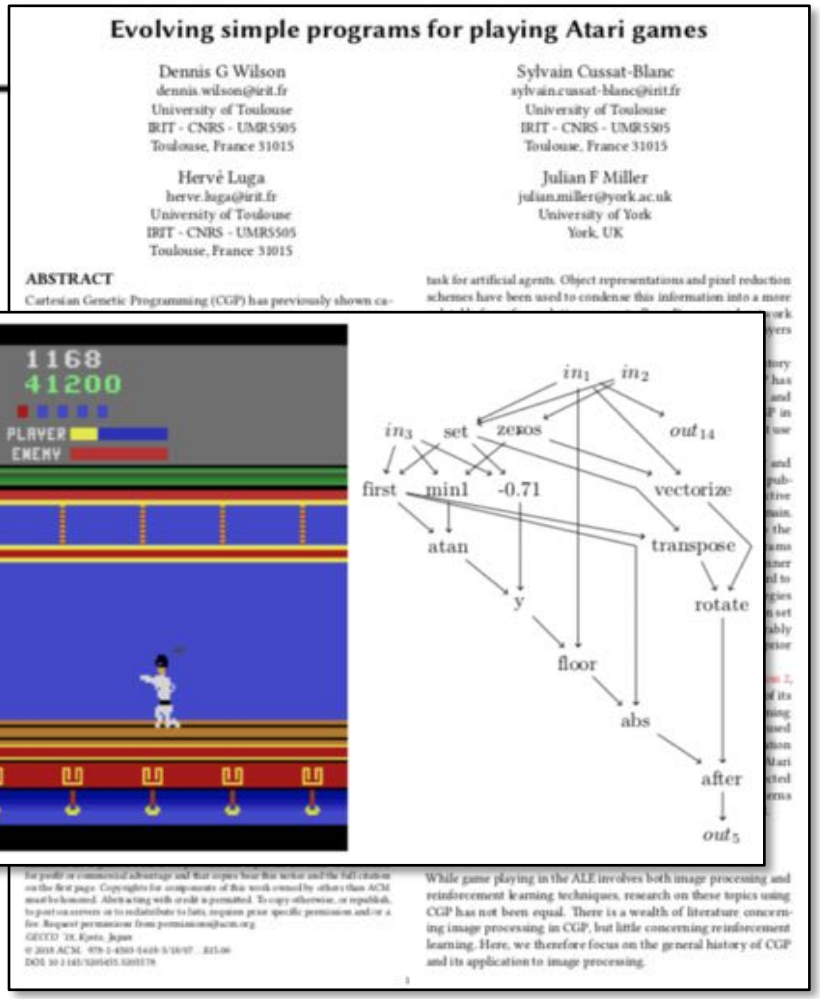
Intelligent Machines

Evolutionary algorithm outperforms deep-learning machines at video games

Neural networks have garnered all the headlines, but a much more powerful approach is waiting in the wings.

by Emerging Technology from the arXiv July 18, 2018

With all the excitement over neural networks and deep-learning techniques, it's easy to imagine that the world of computer science consists of little else. Neural networks, after all, have begun to outperform humans in tasks such as object and face recognition and in games such as chess, Go, and various arcade video games.



Evolving simple programs for playing Atari games

Dennis G Wilson
dennis.wilson@irit.fr
University of Toulouse
IRIT - CNRS - UMR5505
Toulouse, France 31015

Sylvain Cussat-Blanc
sylvain.cussat-blanc@irit.fr
University of Toulouse
IRIT - CNRS - UMR5505
Toulouse, France 31015

Hervé Luga
herve.luga@irit.fr
University of Toulouse
IRIT - CNRS - UMR5505
Toulouse, France 31015

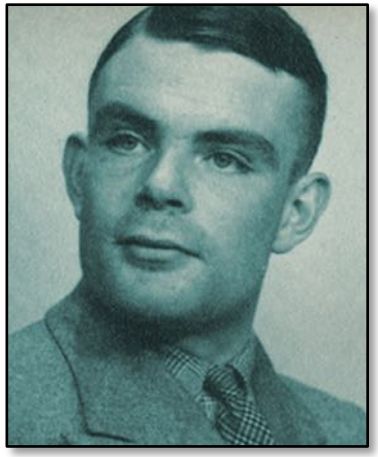
Julian F Miller
julian.miller@york.ac.uk
University of York
York, UK

ABSTRACT

Any Questions?

Evolving in real languages?

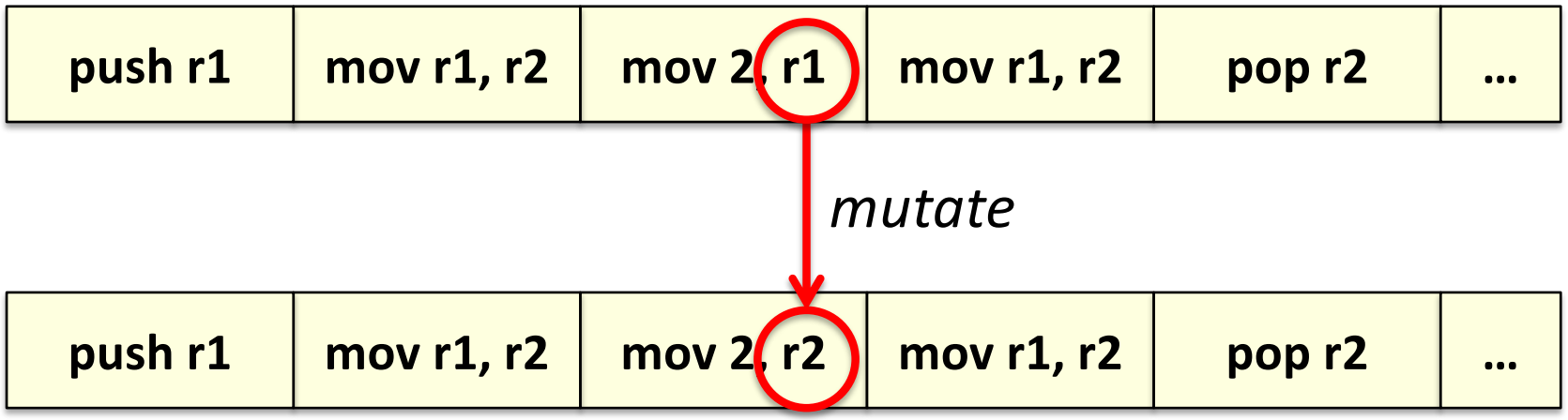
- Tree GP doesn't use standard languages
  - So it requires special interpreters
  - Challenging to integrate with existing code
  - Typically not Turing-complete
  - Language features are rather *ad hoc*



- Other approaches do use conventional languages
  - Machine languages (e.g. x86 assembler): **Linear GP**
  - Imperative languages (e.g. C): **Grammatical evolution**
  - Functional languages (e.g. Haskell): PolyGP
  - Logic languages (e.g. Prolog): DCTG-GP

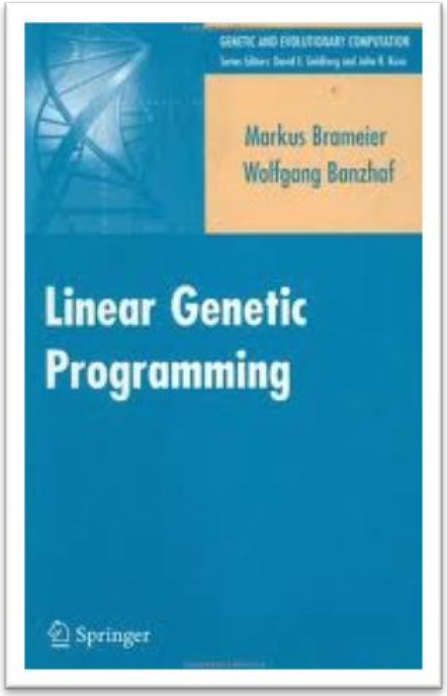
Linear GP

- Evolves a **list** of machine language instructions
  - So, it works a lot like a genetic algorithm
  - Variation operators are less likely to break **syntax**, since low-level languages usually have much simpler syntax
  - There's often no need for an interpreter or a compiler, meaning that it is **fast**



Linear GP

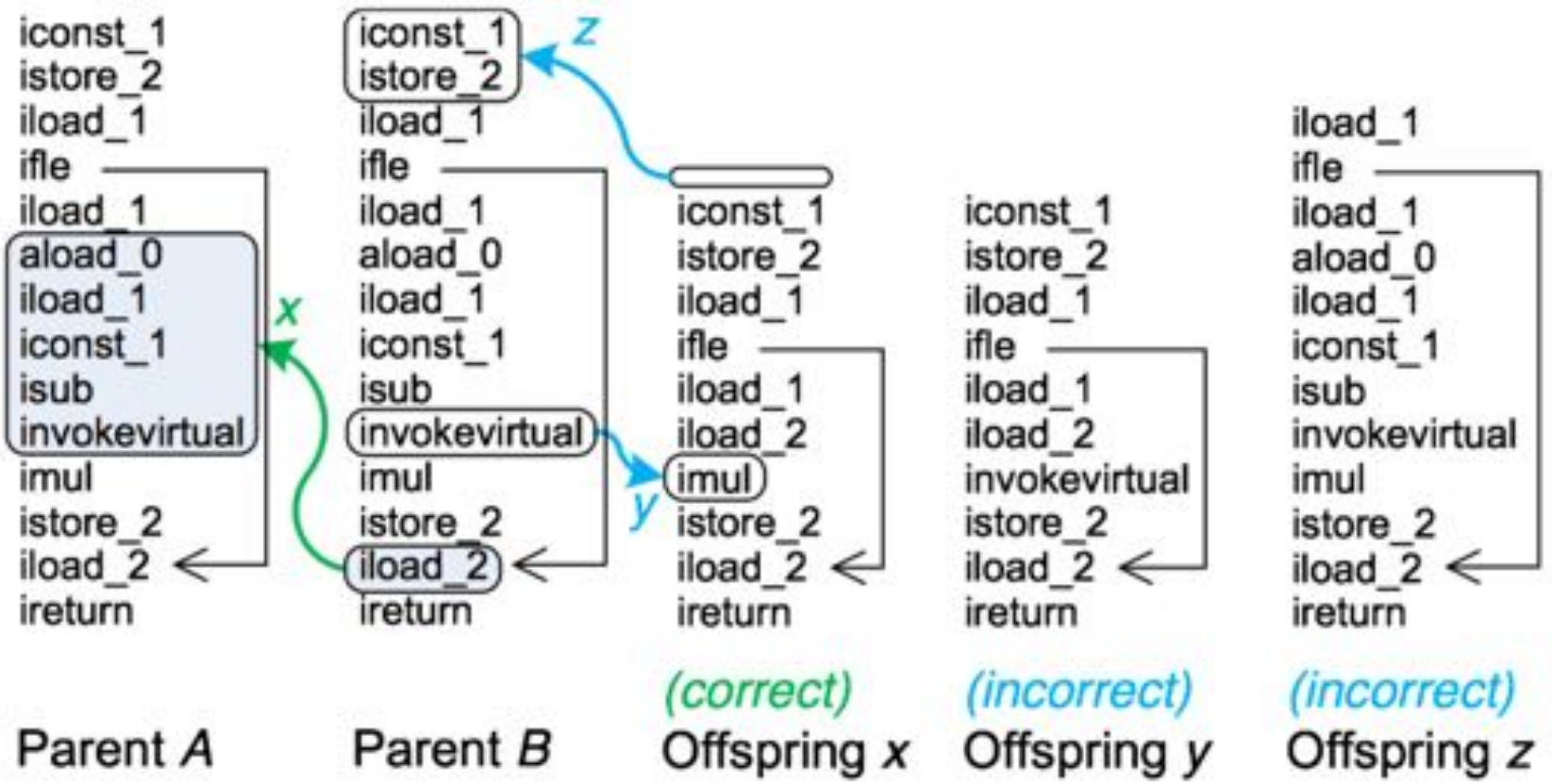
- Languages that have been targeted include:
  - Sun SPARC assembler (RISC)
  - Intel i386 assembler (CISC)
  - Java bytecode
  - CUDA instructions
  - Novel hardware languages
  - GP-specific languages, e.g. Push



http://link.springer.com/book/10.1007%2F978-0-387-31030-5

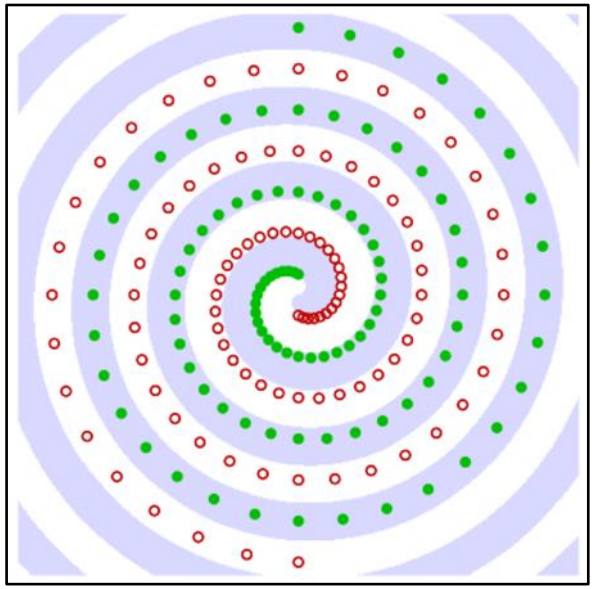
Example

- FINCH [Orlov & Sipper 2010]
  - Evolves programs in Java bytecode. Unlike many low-level languages, this is typed and does require special operators:



Example

- Evolved bytecode programs can be converted to Java
  - e.g. Koza's intertwined spirals problem, which involves finding a classifier that can distinguish the points from two spirals:



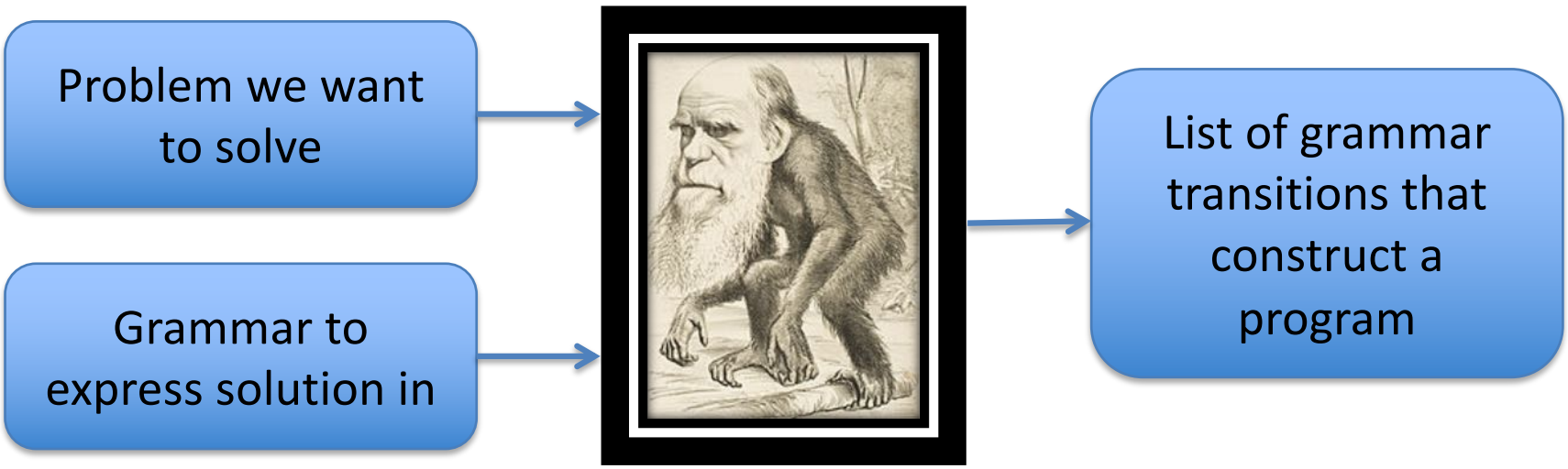
http://finch.cs.bgu.ac.il/

```
boolean isFirst(double x, double y) {
  double a, b, c, e;
  a = Math.hypot(x, y);  e = y;
  c = Math.atan2(y, b = x) +
    -(b = Math.atan2(a, -a))
    * (c = a + a) * (b + (c = b));
  e = -b * Math.sin(c);
  if (e < -0.005612648701876277) {
    b = Math.atan2(a, -a);
    b = Math.atan2(a * c + b, x);  b = x;
    return false;
  }
  else
    return true;
}
```

Any Questions?

Grammatical Evolution (GE)

- Evolves lists of grammar transitions [Ryan 1998]
  - Programming language is expressed by a grammar
  - Described in Backus Naur form (see next slide)
  - GE can then evolve any program in that grammar



Grammatical Evolution (GE)


```
<exp> ::=    0<exp><op><exp> | 1(<exp><op><exp>) |
                        2<pre-op>(<exp>) | 3<var>
<op> ::=     0AND | 1OR | 2XOR
<pre-op> ::= 0NOT
<var> ::=    0IN1 | 1IN2 | 2IN3
```

Evolved chromosome:

10	18	41	27	6	39	21	47	13	31	19	8

- This grammar (top) describes all valid Boolean expressions that can be constructed from three inputs (IN1-3)
- The list of numbers (bottom) describes a single solution, and can be read as a series of grammar transitions...

Grammatical Evolution (GE)

<exp> ::= **^{0}<exp><op><exp> | ^{1}(<exp><op><exp>) | ^{2}<pre-op>(<exp>) | ^{3}<var>**

<op> ::= ^{0}AND | ^{1}OR | ^{2}XOR

<pre-op> ::= ^{0}NOT

<var> ::= ^{0}IN1 | ^{1}IN2 | ^{2}IN3

Evolved

<exp>

For the first step, start with the most general term

10	18	41	27	6	39	21	47	13	31	19	8

10 mod 4 = 2

Number of transitions in current grammar rule

Apply transition 2 to the current term

Grammatical Evolution (GE)


```
<exp> ::=    0<exp><op><exp> | 1(<exp><op><exp>) |
                    2<pre-op>(<exp>) | 3<var>
<op> ::=     0AND | 1OR | 2XOR
<pre-op> ::= 0NOT
<var> ::=    0IN1 | 1IN2 | 2IN3
```

Evolved

For the first step, start with the most general term

<exp>

<pre-op>(<exp>)

10	18	41	27	6	39	21	47	13	31	19	8

10 mod 4 = 2

Number of transitions in current grammar rule

Apply transition 2 to the current term

Grammatical Evolution (GE)


```
<exp> ::=    0<exp><op><exp> | 1(<exp><op><exp>) |
                    2<pre-op>(<exp>) | 3<var>
<op> ::=     0AND | 1OR | 2XOR
<pre-op> ::= 0NOT
<var> ::=    0IN1 | 1IN2 | 2IN3
```

Evolved

<exp>

<pre-op>(<exp>)

At each step, match the left-most unmatched term

10	18	41	27	6	39	21	47	13	31	19	8

18 mod 1 = 0

Number of transitions in current grammar rule

Apply transition 0 to the current term

Grammatical Evolution (GE)


```
<exp> ::=    0<exp><op><exp> | 1(<exp><op><exp>) |
                    2<pre-op>(<exp>) | 3<var>
<op> ::=     0AND | 1OR | 2XOR
<pre-op> ::= 0NOT
<var> ::=    0IN1 | 1IN2 | 2IN3
```

Evolved

10	18	41	27	6	39	21	47	13	31	19	8

At each step, match the left-most unmatched term

<exp>
<pre-op>(<exp>)
NOT(<exp>)

18 mod 1 = 0

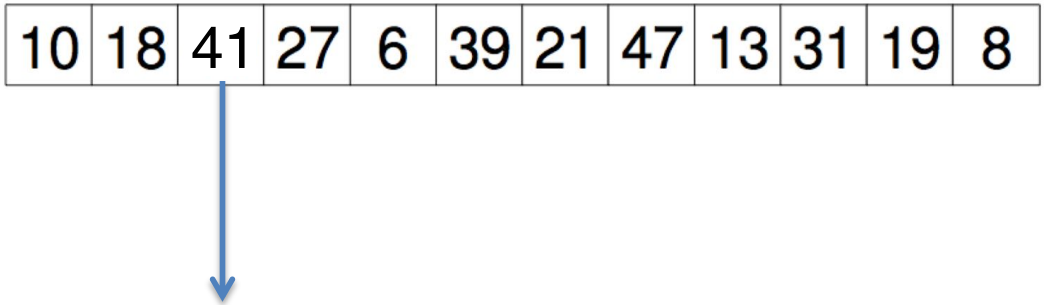
Number of transitions in current grammar rule

Apply transition 0 to the current term

Grammatical Evolution (GE)


```
<exp> ::=    0<exp><op><exp> | 1(<exp><op><exp>) |
                    2<pre-op>(<exp>) | 3<var>
<op> ::=     0AND | 1OR | 2XOR
<pre-op> ::= 0NOT
<var> ::=    0IN1 | 1IN2 | 2IN3
```

Evolved chromosome:



<exp>
<pre-op>(<exp>)
NOT(**<exp>**)
NOT(**<exp><op><exp>**)

Grammatical Evolution (GE)


```
<exp> ::=    0<exp><op><exp> | 1(<exp><op><exp>) |
                    2<pre-op>(<exp>) | 3<var>
<op> ::=     0AND | 1OR | 2XOR
<pre-op> ::= 0NOT
<var> ::=    0IN1 | 1IN2 | 2IN3
```

Evolved chromosome:

<exp>
<pre-op>(<exp>)
NOT(<exp>)
NOT(**<exp>**<op><exp>)
NOT(**<var>**<op><exp>)

27 mod 4 = 3

Grammatical Evolution (GE)


```
<exp> ::=    0<exp><op><exp> | 1(<exp><op><exp>) |
                    2<pre-op>(<exp>) | 3<var>
<op> ::=     0AND | 1OR | 2XOR
<pre-op> ::= 0NOT
<var> ::=    0IN1 | 1IN2 | 2IN3
```

Evolved chromosome:

<exp>
<pre-op>(<exp>)
NOT(<exp>)
NOT(<exp><op><exp>)
NOT(**<var>**<op><exp>)
NOT(**IN1**<op><exp>)

6 mod 3 = 3

Grammatical Evolution (GE)


```
<exp> ::=    0<exp><op><exp> | 1(<exp><op><exp>) |
                    2<pre-op>(<exp>) | 3<var>
<op> ::=     0AND | 1OR | 2XOR
<pre-op> ::= 0NOT
<var> ::=    0IN1 | 1IN2 | 2IN3
```

Evolved chromosome:

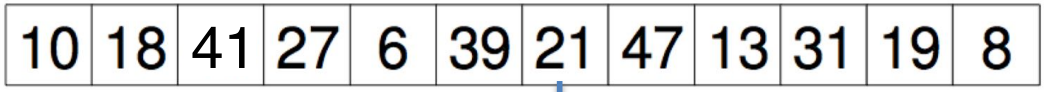
39 mod 3 = 0

<exp>
<pre-op>(<exp>)
NOT(<exp>)
NOT(<exp><op><exp>)
NOT(<var><op><exp>)
NOT(IN1**<op>**<exp>)
NOT(IN1 **AND** <exp>)

Grammatical Evolution (GE)


```
<exp> ::=    0<exp><op><exp> | 1(<exp><op><exp>) |
                    2<pre-op>(<exp>) | 3<var>
<op> ::=     0AND | 1OR | 2XOR
<pre-op> ::= 0NOT
<var> ::=    0IN1 | 1IN2 | 2IN3
```

Evolved chromosome:



21 mod 4 = 1

<pre-op>(<exp>)
NOT(<exp>)
NOT(<exp><op><exp>)
NOT(<var><op><exp>)
NOT(IN1<op><exp>)
NOT(IN1 AND **<exp>**)
NOT(IN1 AND **(<exp><op><exp>)**)

Grammatical Evolution (GE)


```
<exp> ::=    0<exp><op><exp> | 1(<exp><op><exp>) |
                        2<pre-op>(<exp>) | 3<var>
<op> ::=     0AND | 1OR | 2XOR
<pre-op> ::= 0NOT
<var> ::=    0IN1 | 1IN2 | 2IN3
```

Evolved chromosome:

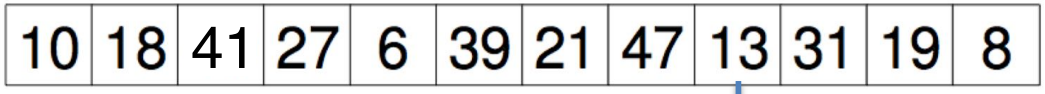
47 mod 4 = 3

NOT(<exp>)
NOT(<exp><op><exp>)
NOT(<var><op><exp>)
NOT(IN1<op><exp>)
NOT(IN1 AND <exp>)
NOT(IN1 AND (**<exp>**<op><exp>))
NOT(IN1 AND (**<var>**<op><exp>))

Grammatical Evolution (GE)


```
<exp> ::=    0<exp><op><exp> | 1(<exp><op><exp>) |
                    2<pre-op>(<exp>) | 3<var>
<op> ::=     0AND | 1OR | 2XOR
<pre-op> ::= 0NOT
<var> ::=    0IN1 | 1IN2 | 2IN3
```

Evolved chromosome:



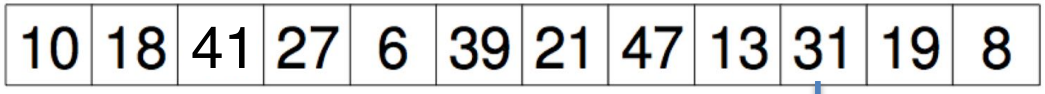
13 mod 3 = 1

NOT(<exp><op><exp>)
NOT(<var><op><exp>)
NOT(IN1<op><exp>)
NOT(IN1 AND <exp>)
NOT(IN1 AND (<exp><op><exp>))
NOT(IN1 AND (**<var>**<op><exp>))
NOT(IN1 AND (**IN2** <op><exp>))

Grammatical Evolution (GE)


```
<exp> ::=     0<exp><op><exp> | 1(<exp><op><exp>) |
                        2<pre-op>(<exp>) | 3<var>
<op> ::=      0AND | 1OR | 2XOR
<pre-op> ::= 0NOT
<var> ::=     0IN1 | 1IN2 | 2IN3
```

Evolved chromosome:



31 mod 3 = 1

NOT(<var><op><exp>)
NOT(IN1<op><exp>)
NOT(IN1 AND <exp>)
NOT(IN1 AND (<exp><op><exp>))
NOT(IN1 AND (<var><op><exp>))
NOT(IN1 AND (IN2 **<op>**<exp>))
NOT(IN1 AND (IN2 **OR** <exp>))

Grammatical Evolution (GE)


```
<exp> ::=    0<exp><op><exp> | 1(<exp><op><exp>) |
                    2<pre-op>(<exp>) | 3<var>
<op> ::=     0AND | 1OR | 2XOR
<pre-op> ::= 0NOT
<var> ::=    0IN1 | 1IN2 | 2IN3
```

Evolved chromosome:

19 mod 4 = 3

NOT(IN1<op><exp>)
NOT(IN1 AND <exp>)
NOT(IN1 AND (<exp><op><exp>))
NOT(IN1 AND (<var><op><exp>))
NOT(IN1 AND (IN2 <op><exp>))
NOT(IN1 AND (IN2 OR **<exp>**))
NOT(IN1 AND (IN2 OR **<var>**))

Grammatical Evolution (GE)


```
<exp> ::=    0<exp><op><exp> | 1(<exp><op><exp>) |
                    2<pre-op>(<exp>) | 3<var>
<op> ::=     0AND | 1OR | 2XOR
<pre-op> ::= 0NOT
<var> ::=    0IN1 | 1IN2 | 2IN3
```

Evolved chromosome:

NOT(IN1 AND <exp>)
NOT(IN1 AND (<exp><op><exp>))
NOT(IN1 AND (<var><op><exp>))
NOT(IN1 AND (IN2 <op><exp>))
NOT(IN1 AND (IN2 OR <exp>))
NOT(IN1 AND (IN2 OR **<var>**))
NOT(IN1 AND (IN2 OR **IN3**))

8 mod 3 = 2

Grammatical Evolution (GE)


```
<exp> ::=    0<exp><op><exp> | 1(<exp><op><exp>) |
                        2<pre-op>(<exp>) | 3<var>
<op> ::=     0AND | 1OR | 2XOR
<pre-op> ::= 0NOT
<var> ::=    0IN1 | 1IN2 | 2IN3
```

Evolved chromosome:

10	18	41	27	6	39	21	47	13	31	19	8

NOT(IN1 AND (IN2 OR IN3))

- If there are **more** numbers than are needed, then the extra ones towards the right of the list are ignored.
- If there are **less** numbers than are needed, the convention is to start again at the beginning.

Grammatical Evolution (GE)


```
<exp> ::=    0<exp><op><exp> | 1(<exp><op><exp>) |
                        2<pre-op>(<exp>) | 3<var>
<op> ::=     0AND | 1OR | 2XOR
<pre-op> ::= 0NOT
<var> ::=    0IN1 | 1IN2 | 2IN3
```

Evolved chromosome:

10	18	41	27	6	39	21	47	13	31	19	8

NOT(IN1 AND (IN2 OR IN3))

- Any changes made to the evolved chromosome will still generate a valid program (a nice feature of GE).

Grammatical Evolution (GE)


```
<exp> ::=    0<exp><op><exp> | 1(<exp><op><exp>) |
                        2<pre-op>(<exp>) | 3<var>
<op> ::=     0AND | 1OR | 2XOR
<pre-op> ::= 0NOT
<var> ::=    0IN1 | 1IN2 | 2IN3
```

Evolved chromosome:

11	18	41	27	6	39	21	47	13	31	19	8

IN1

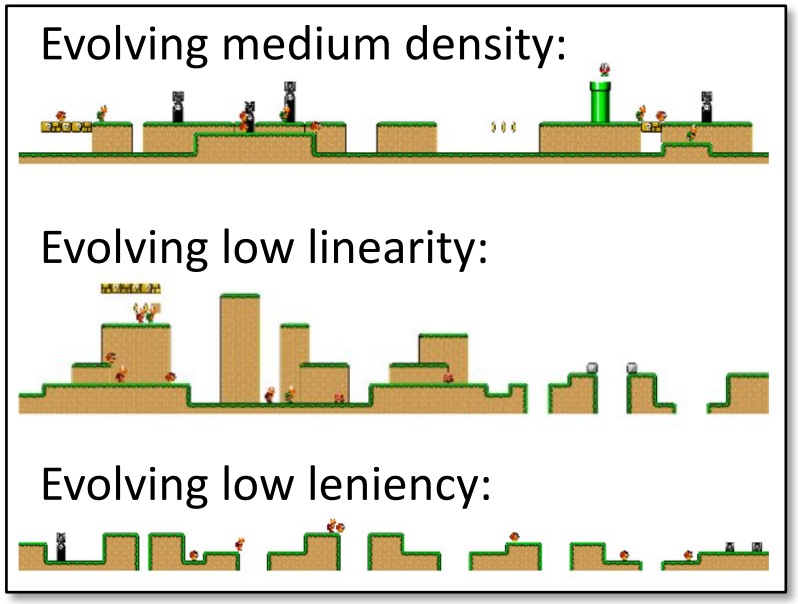
- Although changes made to numbers at the beginning of the list will lead to much larger changes.

Real World Example

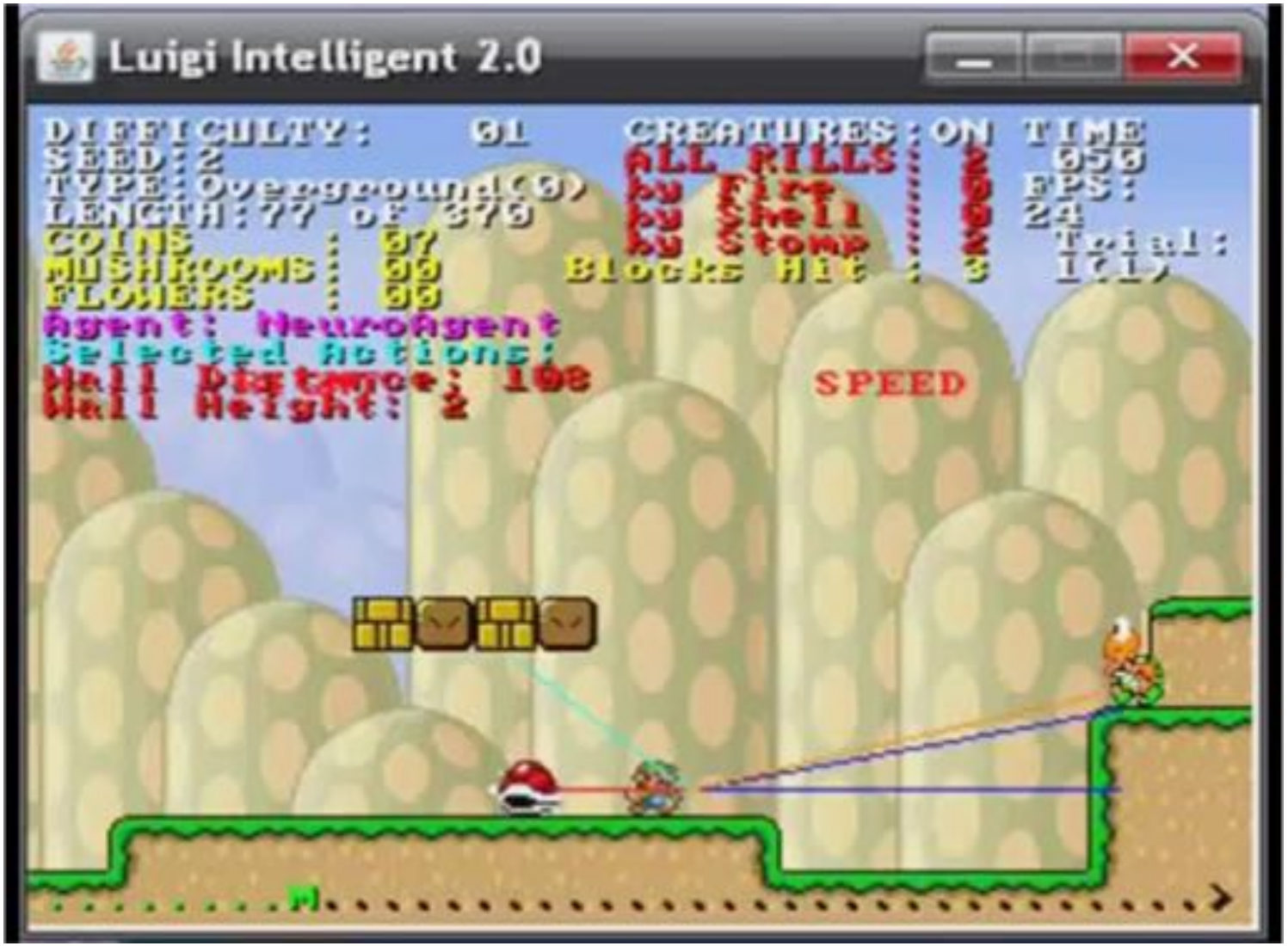
HERIOT WATT UNIVERSITY

- Evolving Super Mario levels [Shaker et al. 2012]

```
<level> ::= <chunks>  <enemy>
<chunks> ::= <chunk> |<chunk> <chunks>
<chunk> ::= gap(<x>,<y>, <w_g>,<w_before>,<w_after>)
  | platform(<x>,<y>,<w>)
  | hill(<x>,<y>,<w>)
  | cannon_hill(<x>,<y>,<h>,<w_before>,<w_after>)
  | tube_hill(<x>,<y>,<h>,<w_before>,<w_after>)
  | coin(<x>,<y>,<w_c>)
  | cannon(<x>,<y>,<h>,<w_before>,<w_after>)
  | tube(<x>,<y>,<h>,<w_before>,<w_after>)
  | <boxes>

<boxes> ::=  <box_type> (<x>,<y>)^2 | ...
  | <box_type> (<x>,<y>)^6

<box_type> ::= block_coin | block_powerup
  | rock_coin  | rock_empty

<enemy> ::= (koopa | goompa)(<x>)^2 | ...
  | (koopa | goompa)(<x>)^10
<x> :: = [5..95] <y> ::= [3..5]
```



Evolving Game Controllers



https://www.youtube.com/watch?v=o5vCQwXt0j8

Grammatical Evolution (GE)

- Relatively flexible and expressive
  - Since grammars can be defined for all languages
  - Programs have been evolved in C, for example
- Though modern languages are problematic
  - Complicated syntax and large APIs => large grammars
  - Typically only a subset of a language is used
- Some concerns about evolvability [Rothlauf, 2006]
  - E.g. small mutations can lead to large changes in the program that is generated

Any Questions?

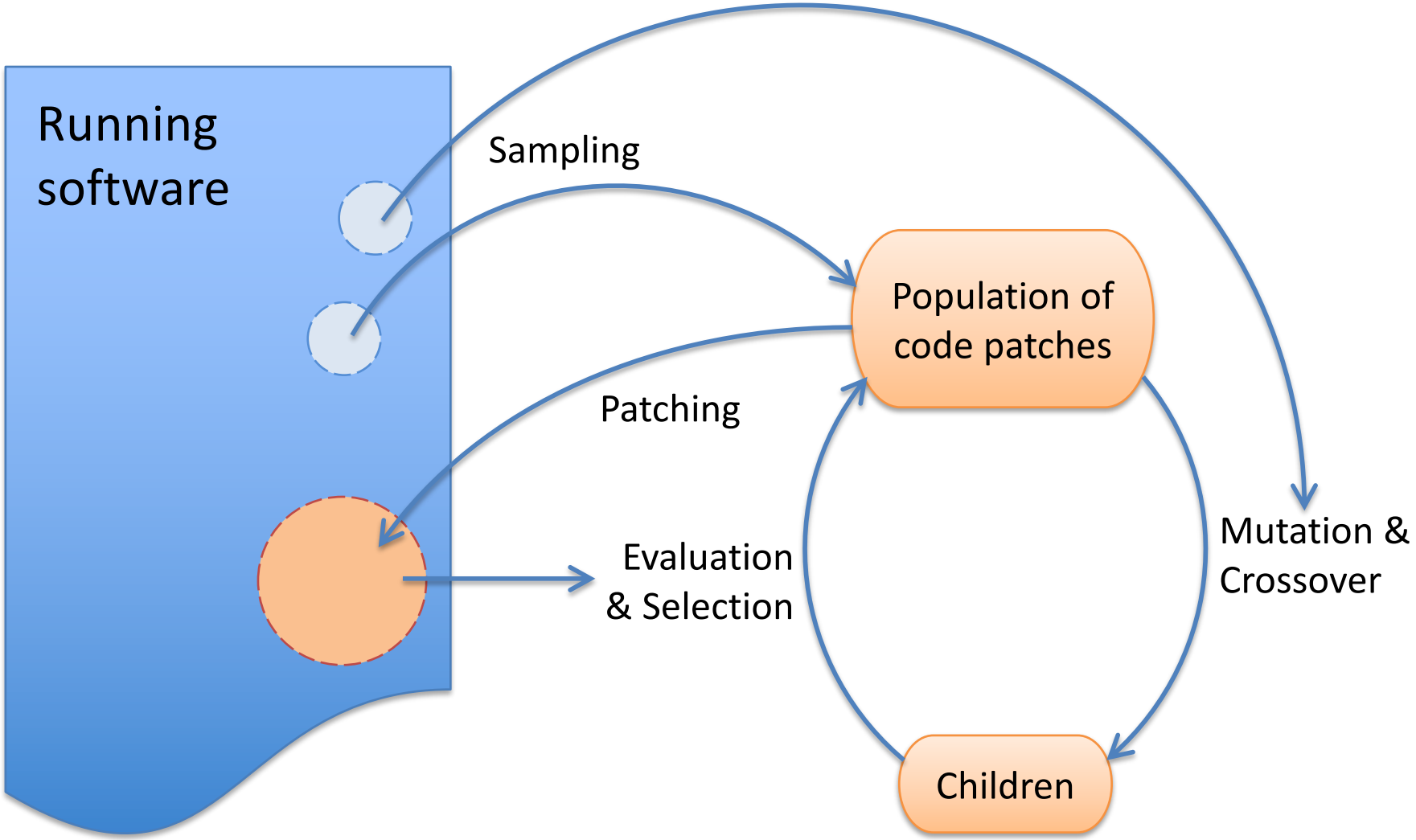
Genetic Improvement (GI)

- The forms of GP discussed so far are all used to evolve small programs from scratch
  - The search space grows exponentially with size
  - So evolving Microsoft Word would take forever
  - And the code wouldn't be maintainable
  - *People are better than GP at writing big programs!*

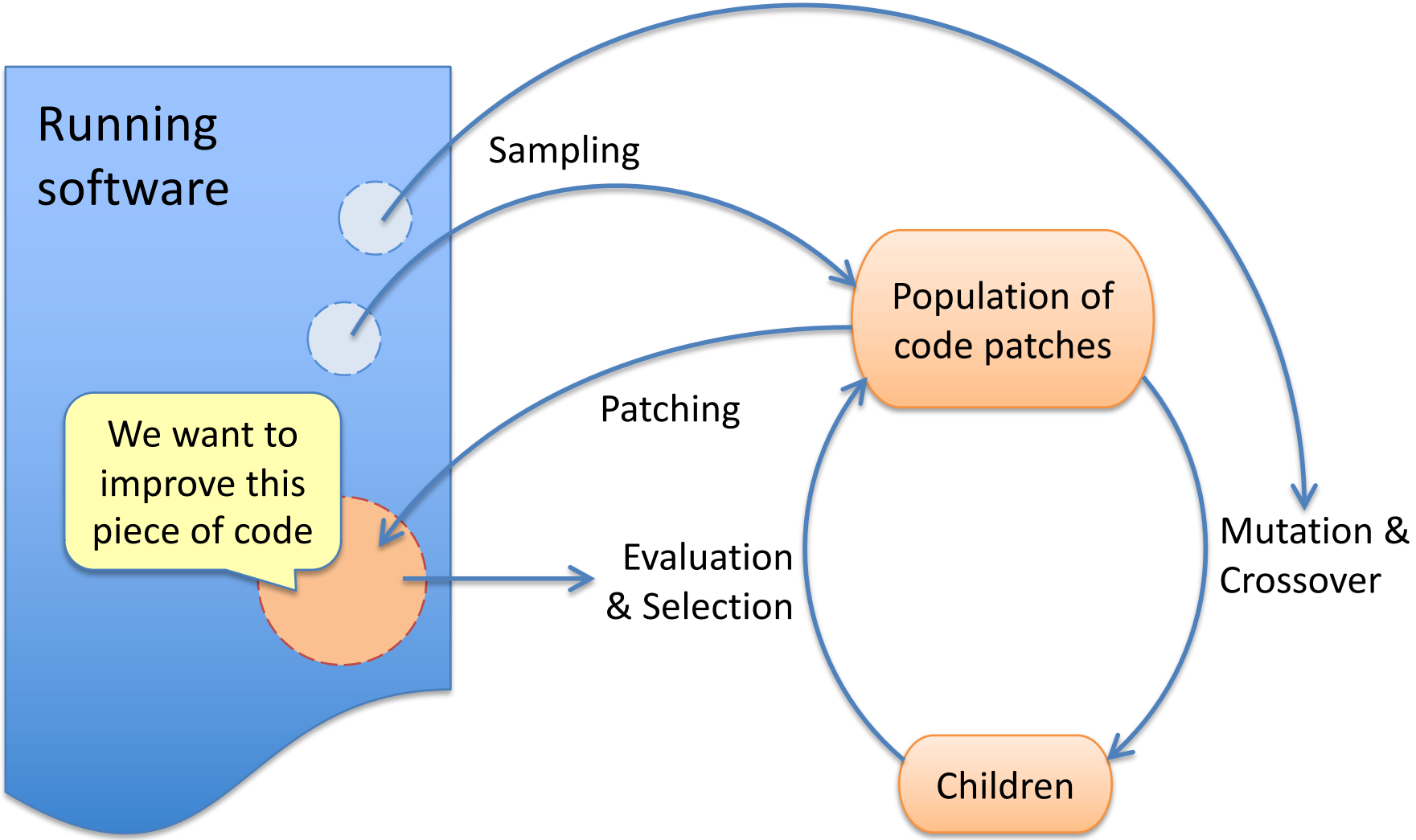
Genetic Improvement (GI)

- GI is a variant of GP that is used to **improve** small parts of existing programs
  - e.g. it improves the efficiency of critical sections
  - e.g. it automatically parallelises critical sections
  - e.g. it translates code into different languages
  - e.g. it fixes bugs
- And it often does this **online**
  - i.e. whilst the target program is running
- Some people see this as GP's **killer app**

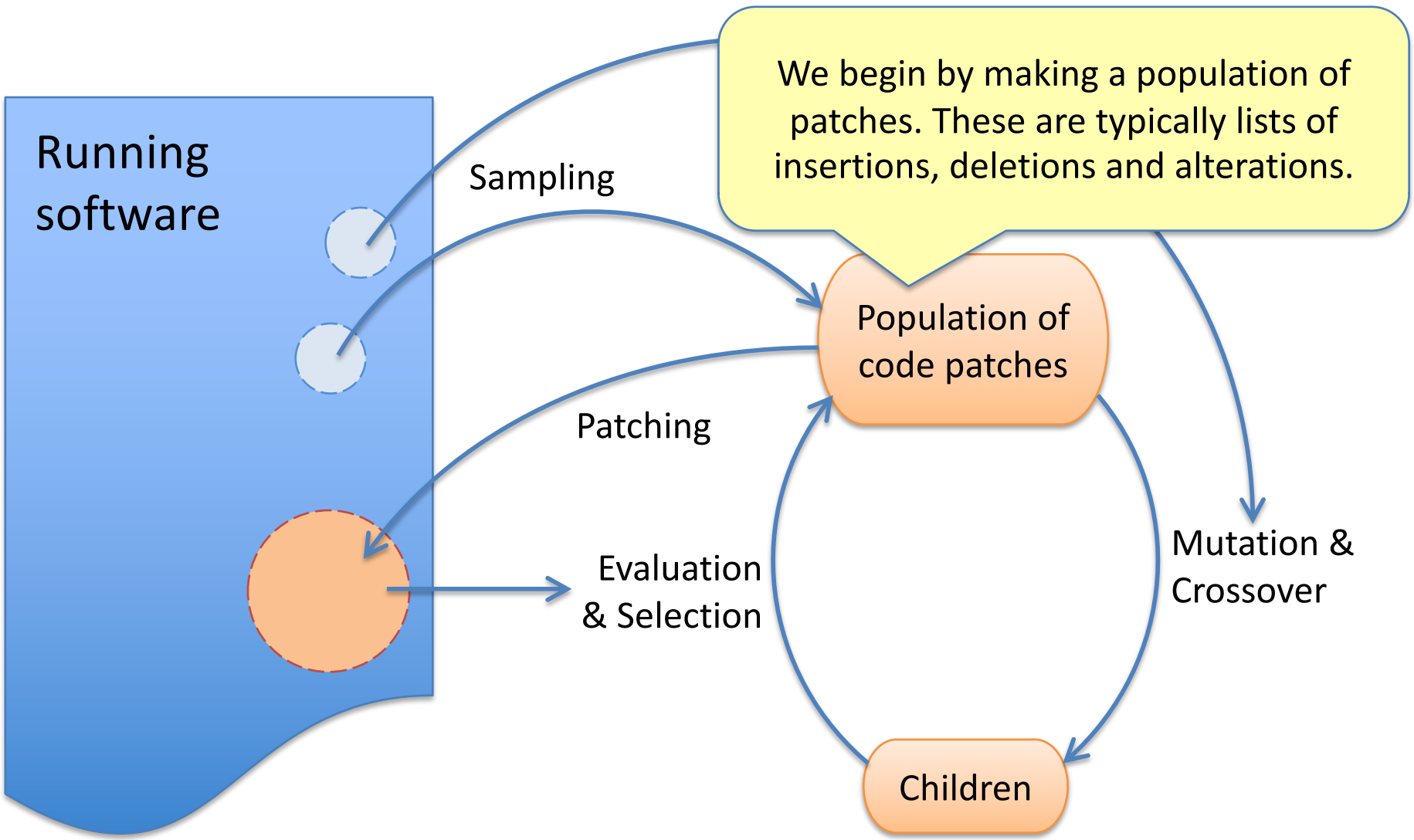
Overview


Running software
Sampling
Population of code patches
Patching
Evaluation & Selection
Mutation & Crossover
Children

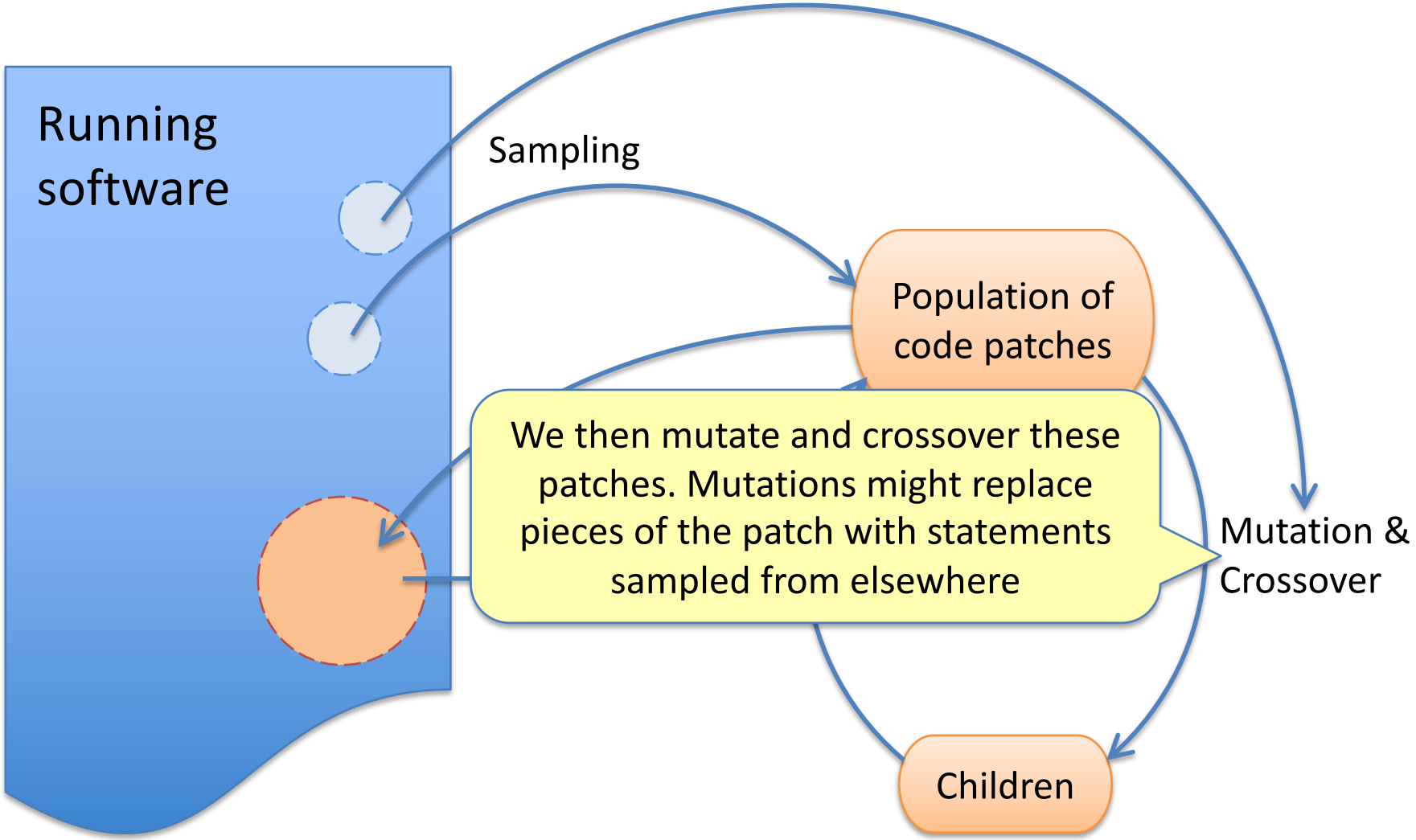
Overview


Running software
We want to improve this piece of code
Sampling
Population of code patches
Patching
Evaluation & Selection
Mutation & Crossover
Children

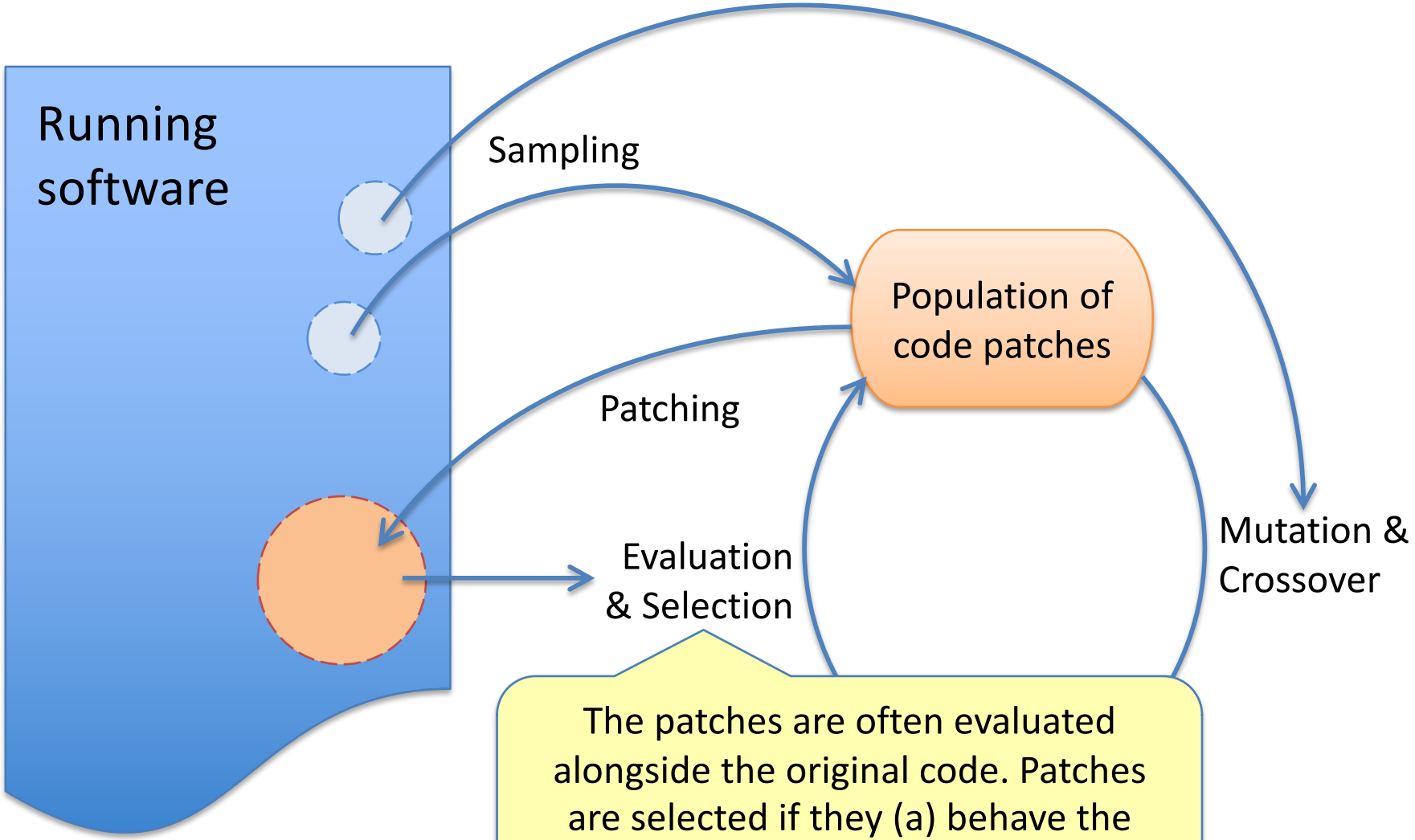
Overview


We begin by making a population of patches. These are typically lists of insertions, deletions and alterations.
Running software
Sampling
Population of code patches
Patching
Evaluation & Selection
Mutation & Crossover
Children

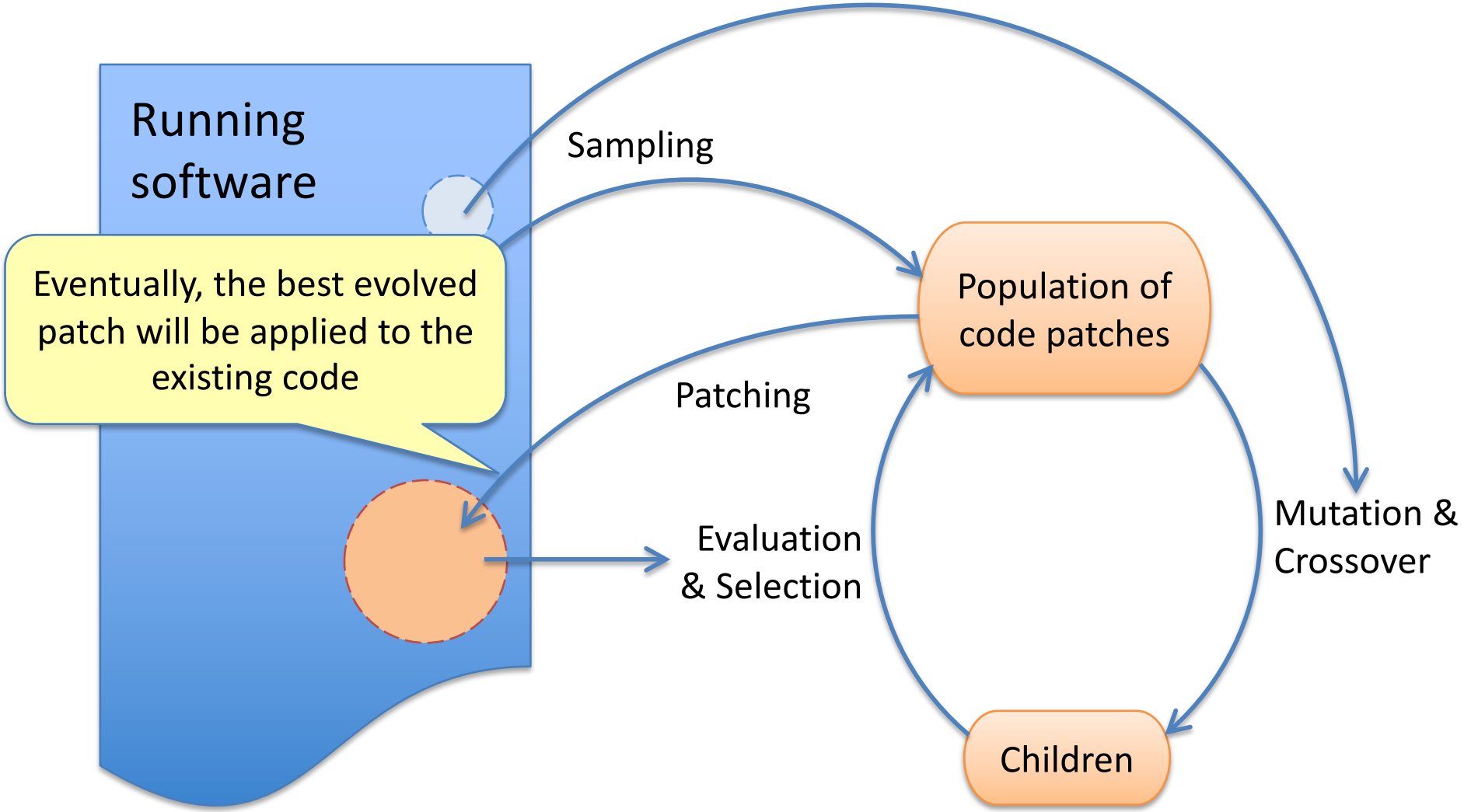
Overview


Running software
Sampling
Population of code patches
We then mutate and crossover these patches. Mutations might replace pieces of the patch with statements sampled from elsewhere
Mutation & Crossover
Children

Overview


Running software
Sampling
Population of code patches
Patching
Evaluation & Selection
Mutation & Crossover
The patches are often evaluated alongside the original code. Patches are selected if they (a) behave the same and (b) improve it in some way

Overview


Running software
Eventually, the best evolved patch will be applied to the existing code
Sampling
Population of code patches
Patching
Evaluation & Selection
Mutation & Crossover
Children

Example: Speeding up

- GI speeded up code in a piece of bioinformatics software, Bowtie2, by a factor of 70 times [Langdon, 2013]
  - http://discovery.ucl.ac.uk/1413298/3/Langdon_2013_ieeeTEC_1.pdf

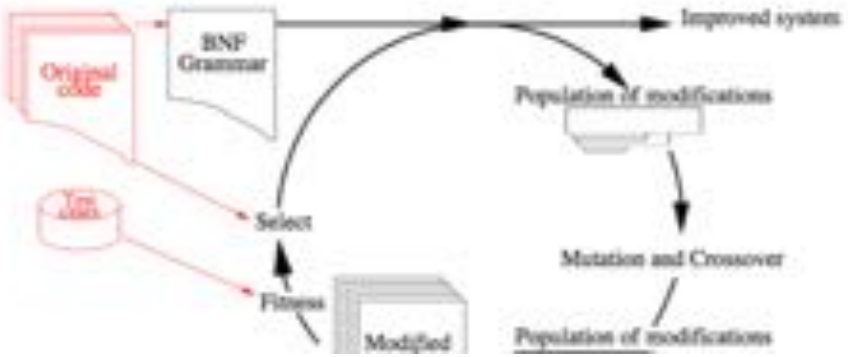
IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION, 1

Optimising Existing Software with Genetic Programming

William B. Langdon and Mark Harman

***Abstract*—We show genetic improvement of programs (GIP) can scale by evolving increased performance in a widely-used and highly complex 50 000 line system. GISMOE found code that is 70 times faster (on average) and yet is at least as good functionally. Indeed it even gives a small semantic gain.**

***Index Terms*—automatic software re-engineering, SBSE, genetic programming, Bowtie2^{GIP}, multiple objective exploration**



Example: Power Usage

- GI used to reduce energy consumption of a piece of software by 25% [Bruce, 2015]
  - http://dl.acm.org/citation.cfm?id=2754752

Reducing Energy Consumption Using Genetic Improvement

Bobby R. Bruce
University College London
London
United Kingdom
r.bruce@cs.ucl.ac.uk

Justyna Petke
University College London
London
United Kingdom
j.petke@ucl.ac.uk

Mark Harman
University College London
London
United Kingdom
mark.harman@ucl.ac.uk

ABSTRACT

Genetic Improvement (GI) is an area of Search Based Software Engineering which seeks to improve software's non-functional properties by treating program code as if it were genetic material which is then evolved to produce more optimal solutions. Hitherto, the majority of focus has been on optimising program's execution time which, though important, is only one of many non-functional targets. The growth in mobile computing, cloud computing infrastructure the world than personal computers [22], each containing a limited store of energy between charges that must be used efficiently. The energy required to run large server clusters has grown considerably in the last decade, estimated to be between 1.1% to 1.5% of global electricity consumption in 2010 [26], putting strain on energy suppliers and the budgets of those responsible for purchasing this energy [7]. The total ICT infrastructure generated 1.9% of global CO_2 emissions in 2011 [5] (larger than the entire United Kingdom estimated

Example: Parallelising

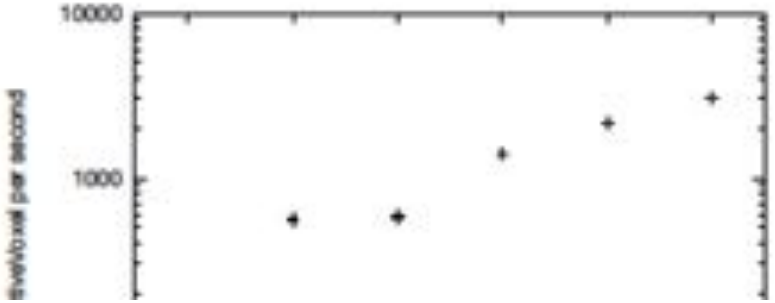
- GI was used to improve the parallelism of a piece of medical image registration software running on a GPU
  - http://dl.acm.org/citation.cfm?id=2598244

Improving 3D Medical Image Registration CUDA Software with Genetic Programming

William B. Langdon, Marc Modat, Justyna Petke, Mark Harman
Dept. of Computer Science, University College London Gower Street, WC1E 6BT, UK
W.Langdon@cs.ucl.ac.uk

ABSTRACT

Genetic Improvement (GI) is shown to optimise, in some cases by more than 35%, a critical component of healthcare industry software across a diverse range of six nVidia graphics processing units (GPUs). GP and other search based software engineering techniques can automatically optimise the current rate limiting CUDA parallel function in the Nifty Reg open source C++ project used to align or reg-



Example: Bug Fixing

- GI was used to automatically patch 22 bugs overnight in a live system left running for 6 months
  - http://dl.acm.org/citation.cfm?id=2598244

Fixing Bugs in Your Sleep: How Genetic Improvement Became an Overnight Success

Saemundur O. Haraldsson*
University of Stirling
Stirling, United Kingdom FK9 4LA
soh@cs.stir.ac.uk

John R. Woodward
University of Stirling
Stirling, United Kingdom FK9 4LA
jrw@cs.stir.ac.uk

Alexander E.I. Brownlee
University of Stirling
Stirling, United Kingdom FK9 4LA
sbr@cs.stir.ac.uk

Kristin Siggeirsdottir
Janus Rehabilitation Centre
Reykjavik, Iceland
kristin@janus.is

ABSTRACT

We present a bespoke live system in commercial use with self-improving capability. During daytime business hours it provides an overview and control for many specialists to simultaneously schedule and observe the rehabilitation process for multiple clients. However in the evening, after the last user logs out, it starts a

1 INTRODUCTION

Genetic Improvement (GI) [38] is a growing area within Search Based Software Engineering (SBSE) [23, 24] which uses computational search methods to improve existing software. Despite its growth within academic research the practical usage of GI has not yet followed. Like with many SBSE applications, the software in-

Genetic Improvement (GI)

- What does a patch look like?
  - This varies, but commonly it contains snippets of source code represented as strings of text
  - i.e. rather than conventional GP representations, such as parse trees
  - Lines of code can be removed, inserted or altered, e.g.
  - REMOVE line 5
    INSERT line 7 "**`vmax = vlo;`**"
    REPLACE line 12, col 5 "**`<`**", "**`<=`**"
    …

Genetic Improvement (GI)

- How does initialisation/mutation work?
  - A common approach is to use **code found elsewhere** in the program, e.g. replace a line of code with one from another method in the same program
  - This works surprisingly often, since programmers use similar code in different parts of a program
  - In principle, code could also be taken **from other programs**, or foraged from places like Stackexchange
  - Another approach is to **generate a grammar** based on existing code, and then use this to generate new code

Genetic Improvement (GI)

- E.g. [Langdon, 2013]

Grammar

```
<bowtie_main_42>        ::= "int main(int argc, const char **argv) {\n"
<bowtie_main_43>        ::= "{Log_count64++;/*29823*/} if" <IF_bowtie_main_43> " {\n"
#"if
<IF_bowtie_main_43>     ::= "(argc > 2 && strcmp(argv[1], \"-A\") == 0)"
<bowtie_main_44>        ::= "const char *file = argv[2];\n"
<bowtie_main_45>        ::= "ifstream in;\n"
<bowtie_main_46>        ::= "" <_bowtie_main_46> "{Log_count64++;/*29826*/}\n"
#other
<_bowtie_main_46>       ::= "in.open(file);"
<bowtie_main_47>        ::= "char buf[4096];\n"
```

Patch

Mutation	Source file	line	type	Original Code	New Code
replaced	bt2_io.cpp	622	for2	`i < offsLenSampled`	`i < this->_nPat`
replaced	sa_rescomb.cpp	50	for2	`i < satup_->offs.size()`	0
disabled	sa_rescomb.cpp	69	for2	`j < satup_->offs.size()`	
replaced	aligner_swsse_ee_u8.cpp	707		`vh = _mm_max_epu8(vh, vf);`	`vmax = vlo;`
deleted	aligner_swsse_ee_u8.cpp	766		`pvFStore += 4;`	
replaced	aligner_swsse_ee_u8.cpp	772		`_mm_store_si128(pvHStore, vh);`	`vh = _mm_max_epu8(vh, vf);`
deleted	aligner_swsse_ee_u8.cpp	778		`ve = _mm_max_epu8(ve, vh);`	

Genetic Improvement (GI)

- How do we check that a patch behaves the same as the existing code?
  - Typically this requires the program which is being improved to have **unit tests**
  - These check whether the code produces the same output for the same inputs
  - Most mature software systems will have existing suites of unit tests, so this does not usually require extra work

Any Questions?

Summary

Name	Representation	Strengths	Weaknesses
Koza GP	Parse tree of functions and terminals	Well known and widely supported	Does not use real languages, tends to bloat
Cartesian GP (CGP)	Graph/grid of interconnected functions/terminals	Supports implicit reuse and neutral evolution. No bloat.	Does not use real languages, fixed grid size
Linear GP	Vector/list of instructions	Supports low-level languages, relatively fast, less syntax issues	Doesn't work with high-level languages
Grammatical Evolution (GE)	Vector/list of grammar transitions	Supports high-level languages	Some issues with evolvability, limited grammar size
Genetic Improvement (GI)	Patches: lines of code to insert and delete into/from a program	Does not need to evolve everything from scratch	Needs things like unit tests to maintain correctness

Suggested Reading

HERIOT WATT UNIVERSITY

- Read Chapter 7 of "Field Guide to GP", pp 62-68
  - An intro to linear and graphical GP
- Read Saemundur Haraldsson et al's paper
  - *"Fixing Bugs In Your Sleep: How Genetic Improvement Became an Overnight Success"*
  - A nice example of using GI in the real world
  - On Vision

Other things you could try out

- Have a look at Grammatical Evolution
  - `ec.gp.ge` package in ECJ
  - http://cs.gmu.edu/~eclab/projects/ecj/docs/manual/manual.pdf
    - See section 5.3

- Maybe have a look at Linear GP (not supported by ECJ)
  - Discipulus: http://www.rmltech.com (commercial, trial version)
  - Slash/A: https://github.com/arturadib/slash-a

Bibliography

- B. Bruce et al., Reducing Energy Consumption Using Genetic Improvement, Proceedings of GECCO 2015, ACM, https://dl.acm.org/citation.cfm?id=2754752
- W. Langdon et al., Improving 3D medical image registration CUDA software with genetic programming, Proceedings of GECCO 2014, https://dl.acm.org/citation.cfm?id=2598244
- W. Langdon and M. Harman, Optimising Existing Software with Genetic Programming, IEEE Transactions on Evol. Comp. 19(1):118-135, doi:10.1109/TEVC.2013.2281544
- J. Miller and P. Thomson, Cartesian Genetic Programming, 2000, https://link.springer.com/chapter/10.1007/978-3-540-46239-2_9
- M. Orlov and M. Sipper, Flight of the FINCH Through the Java Wilderness, IEEE Transactions on Evolutionary Computation 15(2), 2011 http://finch.cs.bgu.ac.il/pubs/finch.pdf
- F. Rothlauf, M. Oetzel, On the Locality of Grammatical Evolution, 2006, https://link.springer.com/chapter/10.1007%2F11729976_29
- C. Ryan, Grammatical Evolution: Evolving Programs for an Arbitrary Language, EuroGP 1998, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.38.7697
- N. Shaker et al, Evolving Levels for Super Mario Bros Using Grammatical Evolution, CIG2012, http://noorshaker.com/docs/ELGE.pdf

- Note: many of these URLs will only work whilst on the campus network!

Next Lecture

- Multiobjective Evolutionary Algorithms (MOEAs)
  - Widely used to solve challenging real world problems