MATHEW CARR

MSc. Project:

Code Generation Through Genetic Programming

FINAL PRESENTATION

Supervisor: David Jackson

Aims of Project

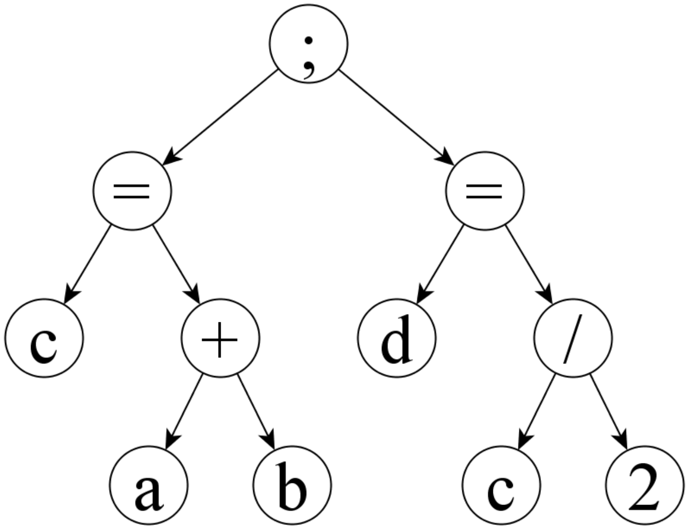
- Investigate methods where Linear Genetic Programming techniques can be applied to achieve or expedite code generation
- Produce software allowing experiments to be conducted

Methods

- ‘Standard’
  - Evolution of a single solution program with the same semantics as the input program
- ‘Incremental’
  - Division of input program into smaller subprograms
  - Evolution of solution programs for each subprogram
  - Concatenation of partial solutions into complete solution

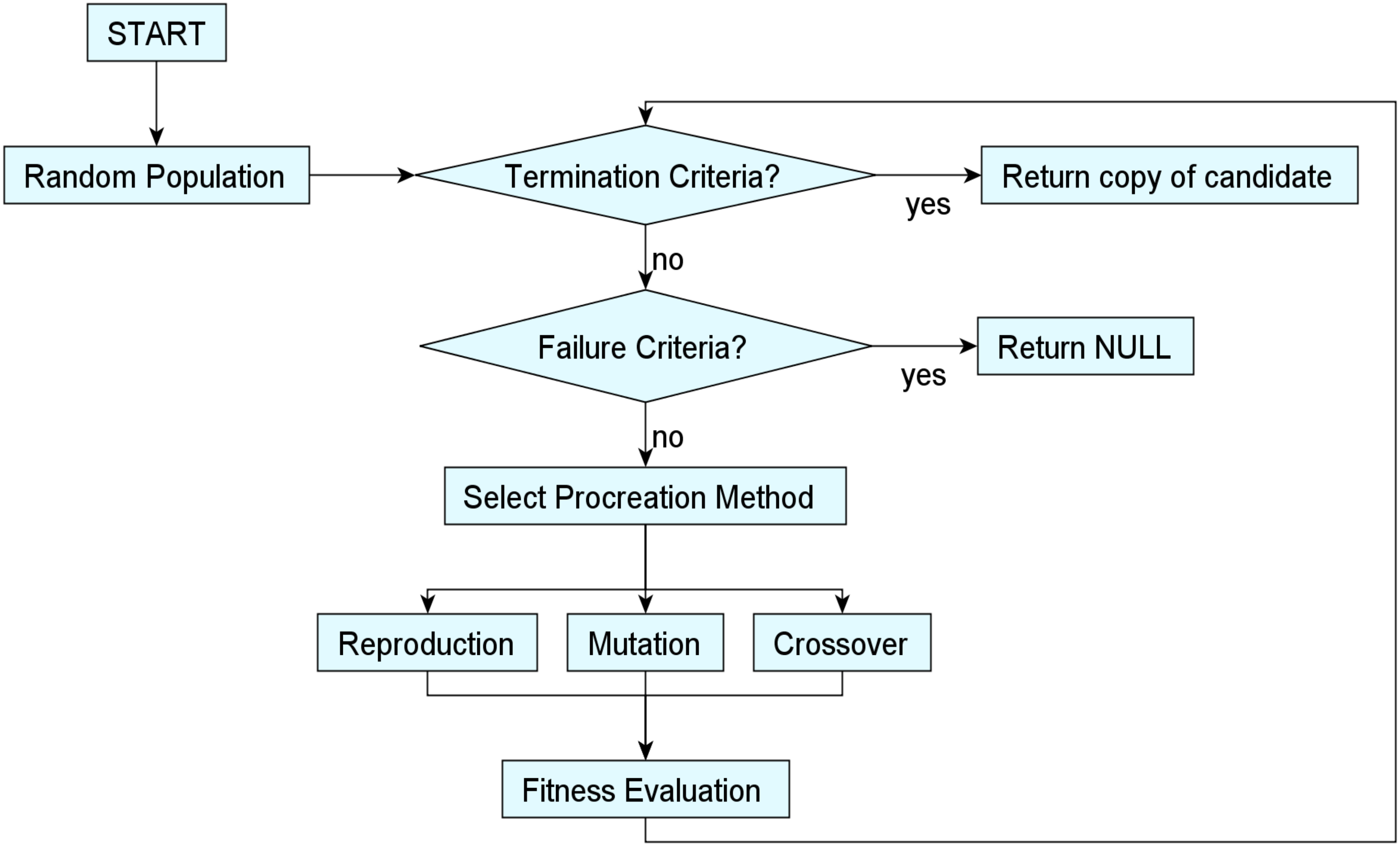
“Calculate the mean of the values of the variables a and b and store the result in variable d”

```
c = a + b;

d = c / 2
```



```
LOADS 0,    a     // load the value of variable a into r0
LOADS 1,    b     // load the value of variable b into r1
ADD   0, 0, 1     // add the values of r0 and r1; store result in r0
LOADV 1,    2     // load the direct value 2 into r1
DIVP  0, 0, 1     // divide value of r0 by that of r1; store result in r0
STORS 0,    d     // store the value of register 0 into variable d
HALT              // end program
```

Evolve Instruction String From Parse Tree



START

Random Population

Termination Criteria?

yes

Return copy of candidate

no

Failure Criteria?

yes

Return NULL

no

Select Procreation Method

Reproduction

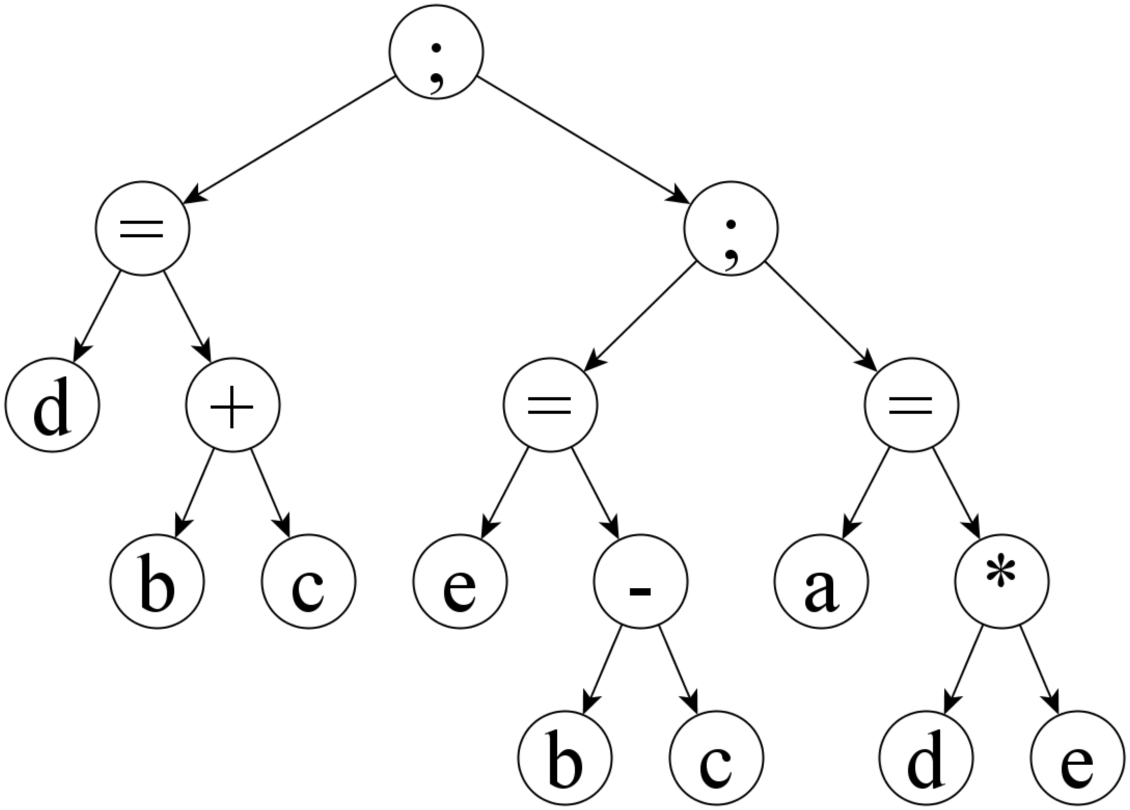
Mutation

Crossover

Fitness Evaluation

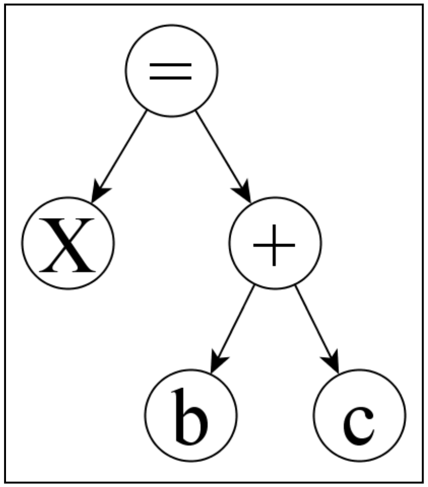
‘Incremental’

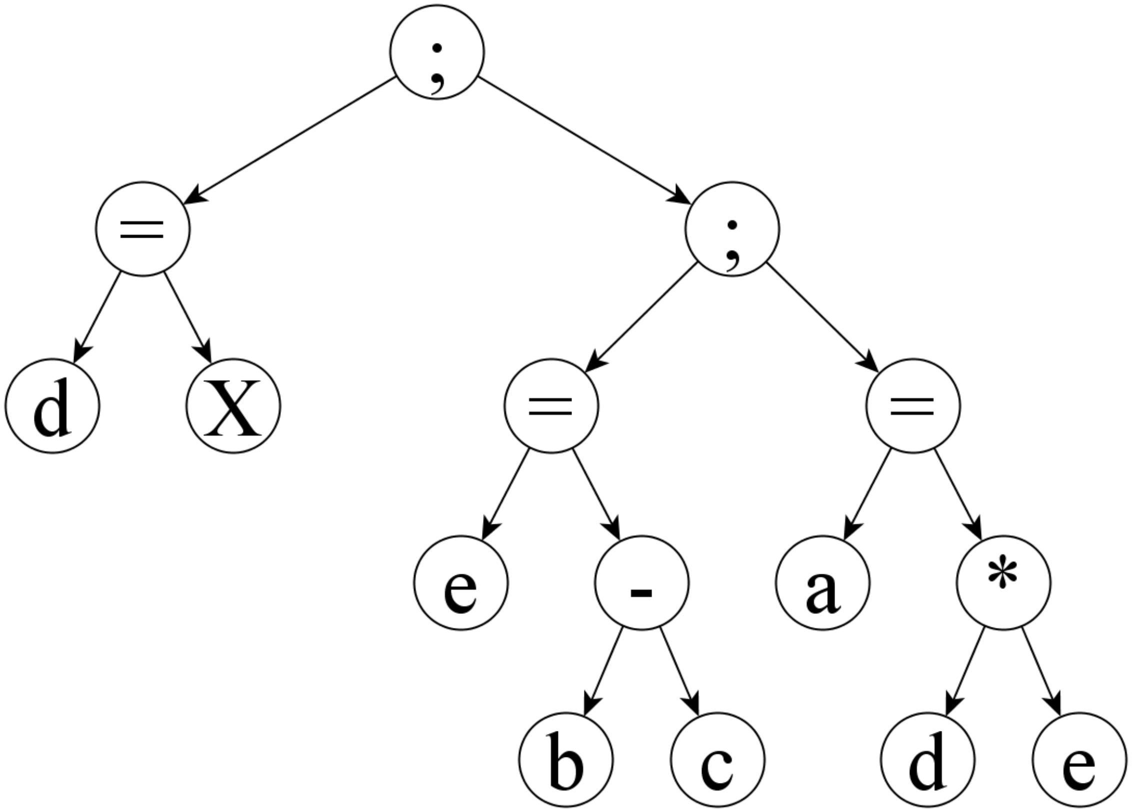
```
d = b + c;

e = b - c;

a = d * e
```



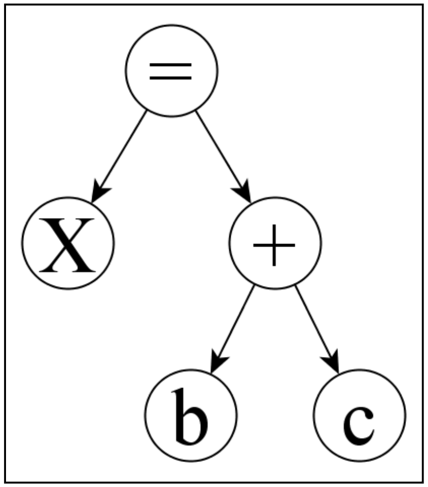
Mathew Carr

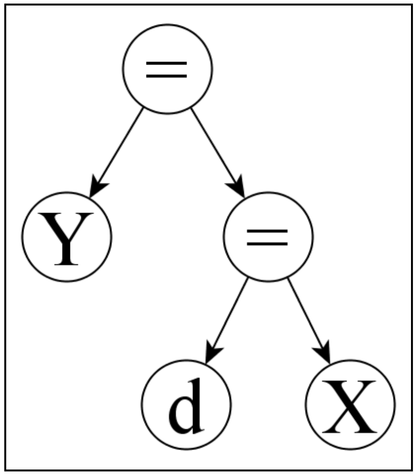
‘Incremental’

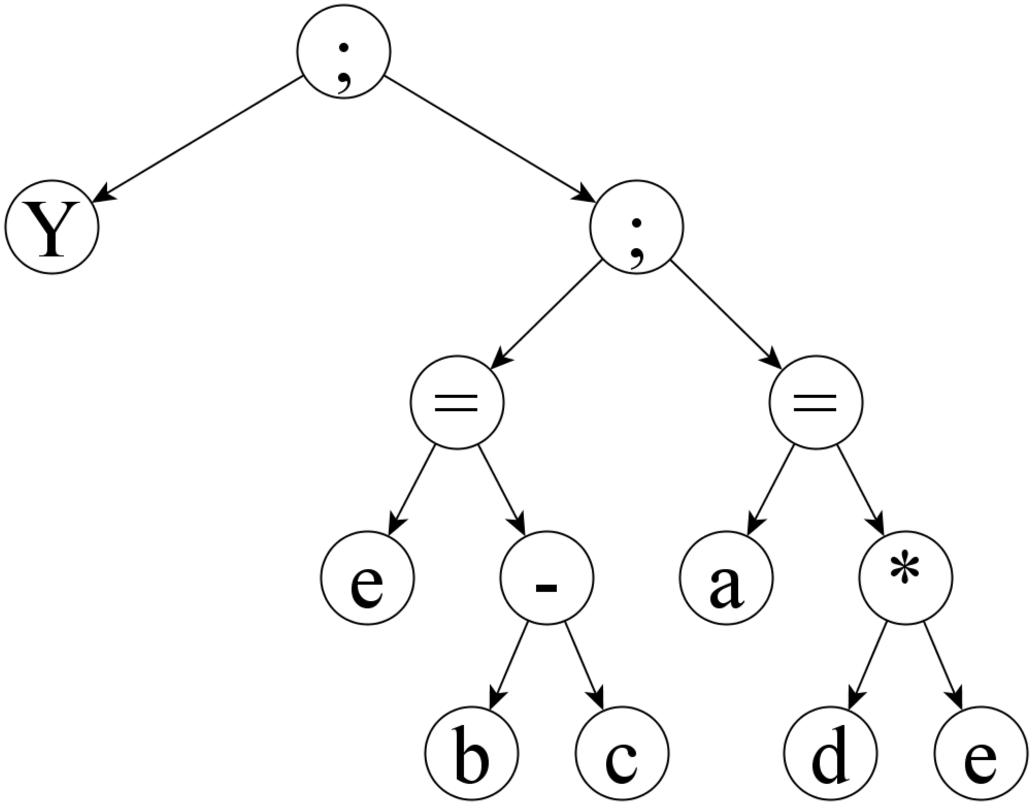




‘Incremental’







Evolutionary Program Refinement

- Final stage of processing after program is found
- Attempt to improve input program using the same LGP operations as before
- Fraction of the initial population is duplicates of the previously evolved solution program
  - Fitness: primarily based on program length
- Terminate after a fixed number of program creations

Metrics Used

- Ten simple source programs:
- ‘Computational Effort’
  - Minimum number of instructions required to produce solution program with 99% probability
- ‘Program Length’
  - Distribution of lengths of solution programs
  - ‘Cost’ of solution program

Computational Effort

- Allows for comparison of apparent difficulty between ‘incremental’ and ‘standard’ methods.
- Higher value indicates more time is needed to produce a solution program using this method

Computational Effort

- Incremental appears to scale linearly with number of internal nodes
  - Only has to solve small programs: fewer goals
  - Small symbol table: fewer possible instructions

- Standard appears to scale exponentially with number of internal nodes
  - Many constraints on what makes a valid program: many goals
  - Many genetic operations act destructively

Program Length

- Standard
- Incremental
- Standard with refinement
- Incremental with refinement

- Non-optimising, tree walking compiler algorithm
- Tree walking compiler algorithm with refine

Program Length

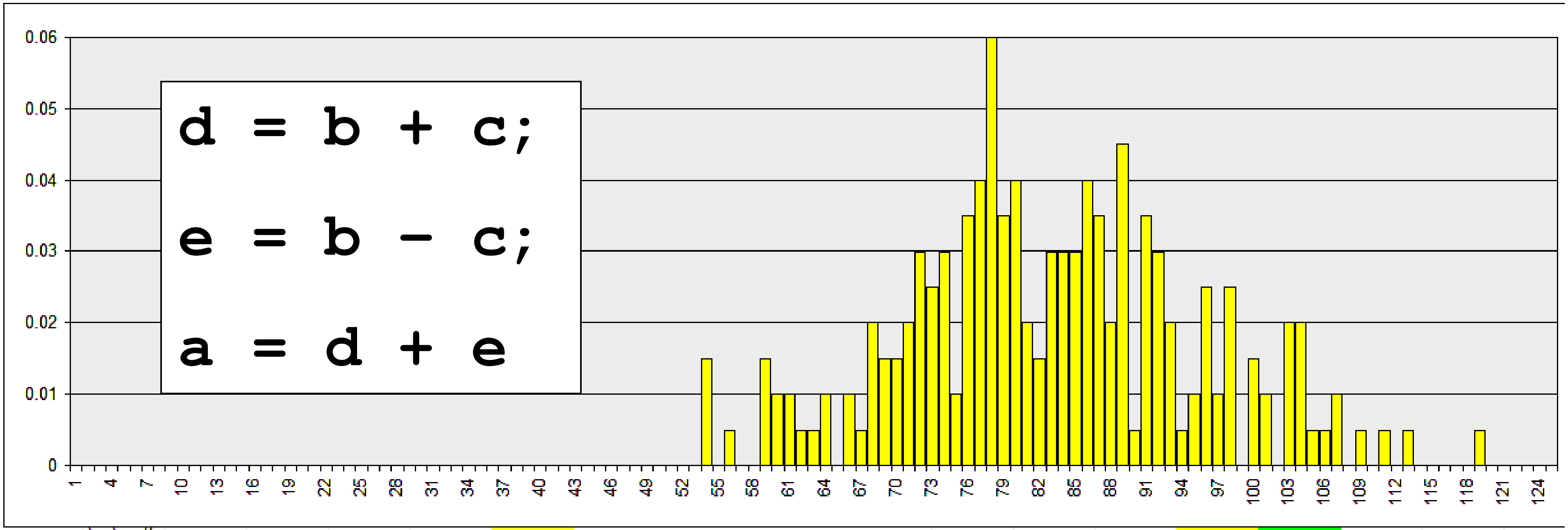
- Standard approach produces shorter programs; half the length of those produced by incremental
  - At a cost of greatly increased computational effort
- Tree walking algorithm produces superior programs under all cases

Program Length, Refinement

- Refinement generally reduces program length by 50%

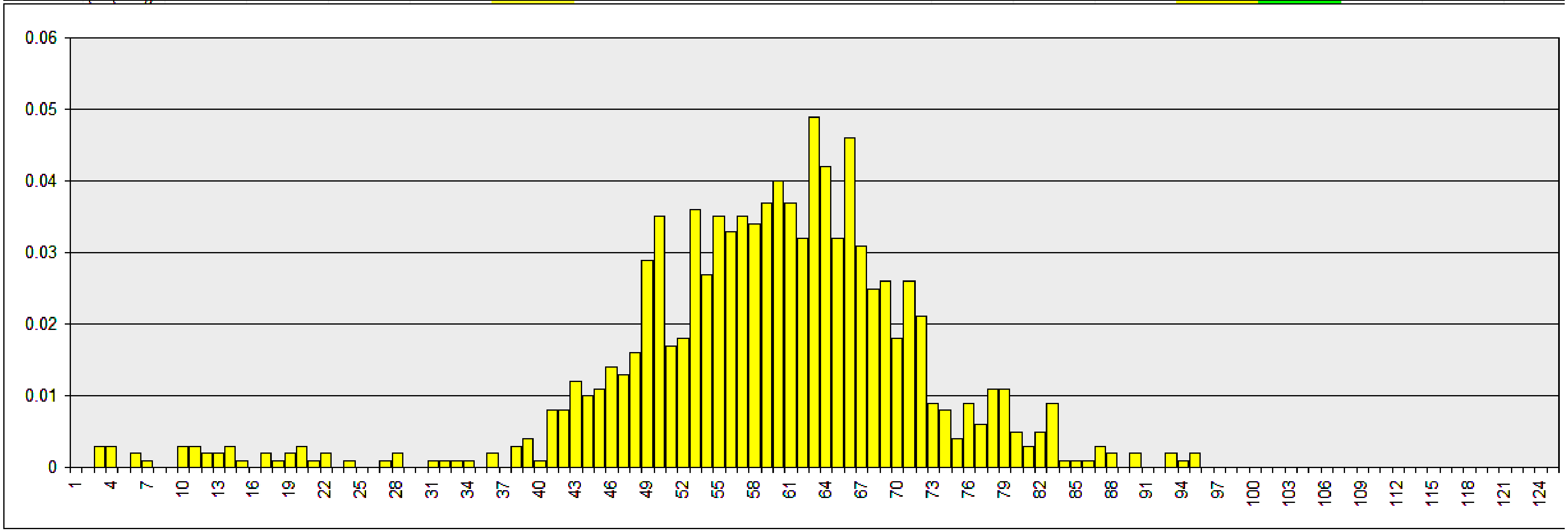
- Programs produced by standard are more easily refined
  - Altering complex programs requires intermediate states with lower fitness

- Capable of producing optimal programs



```
d = b + c;
e = b - c;
a = d + e
```

0.06
0.05
0.04
0.03
0.02
0.01
0
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67 70 73 76 79 82 85 88 91 94 97 100 103 106 109 112 115 118 121 124



0.06
0.05
0.04
0.03
0.02
0.01
0
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67 70 73 76 79 82 85 88 91 94 97 100 103 106 109 112 115 118 121 124

Limitations of Project

- No powerful instructions
  - Trivial translation by tree walking algorithm
- Short programs
  - Few opportunities for optimisation
- Sufficient register file
  - Advantage to tree walking algorithm

Thank you

Any questions?