Fixing bugs in Python programs with Genetic Improvement

Program size and search granularity
Overview of talk

- Developing a GI framework for Python programs
- Search granularity and program size
- Breaking and fixing small Python programs
Motivation

- GI has already been successfully applied to large software, >50K LOC (Langdon et al. & Le Goues et al.)
- Pushing GI to its lower size limit for usefulness
- “The competent programmer hypothesis” for students
- Easier to analyse exactly what the GI is doing
Gl for Python
GI for Python —— Entities of the population

- **Evolving Edit lists**

- **Available edits**
  - Copy, Swap, Delete and Replace

- **Movable code**
  - Whole Lines
  - **Boolean operators**: 'or', 'and', 'not', '<=', '!=', etc.
  - **Mathematical operators**: '+', '*', '-', '%', etc
  - **Incremental operators**: '+=', '*=', '/=', '-='
  - Numerical constants

- **Fitness function**
  - Number of passed test cases
GI for Python ---- Features of the evolution

- The usual customizable properties
  - Population size
  - Number of generations
  - Selection
  - Survival / Elitism
- Offspring entities made with mutation only
  - Grow: Append randomly generated edits
  - Prune: Shorten the list of edits
  - Single edit mutation: Randomly select 1 edit and change it slightly.
GI for Python ---- Features of the evolution

- The usual customizable properties
  - Population size
  - Number of generations
  - Selection
  - Survival / Elitism
- Offspring entities made with **mutation** only
  - Grow: Append randomly generated edits
  - Prune: Shorten the list of edits
  - Single edit mutation: Randomly select 1 edit and change it slightly.
GI for Python ---- Features of the evolution

- The usual customizable properties
  - Population size
  - Number of generations
  - Selection
  - Survival / Elitism
- Offspring entities made with mutation only
  - Grow: Append randomly generated edits
  - Prune: Shorten the list of edits
  - Single edit mutation: Randomly select 1 edit and change it slightly.

<REPLACE, ‘<’, ‘>’, 34, 12>
<REPLACE, ‘2’, ‘1’, 65, 20>
<REPLACE, ‘<’, ‘>’, 34, 12>
GI for Python

Features of the evolution

- The usual customizable properties
  - Population size
  - Number of generations
  - Selection
  - Survival / Elitism

- Offspring entities made with mutation only
  - Grow: Append randomly generated edits
  - Prune: Shorten the list of edits
  - Single edit mutation: Randomly select 1 edit and change it
Search Granularity
Program Size
Search Granularity ----- Experimental setup

Movable code

- **Random** line edits
- **Like for like** line edits
- **Change operators**: math, boolean and incremental.

Step size (mutation choices)

- **Grow and prune** only (variable size)
- **Single edit** mutations and Grow (single edit growth)
- **Both above**

<table>
<thead>
<tr>
<th>Movable code</th>
<th>Step size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>All available</td>
</tr>
<tr>
<td>Random lines</td>
<td></td>
</tr>
<tr>
<td>Like for like lines</td>
<td>X</td>
</tr>
<tr>
<td>Operators and numbers</td>
<td>X</td>
</tr>
</tbody>
</table>
Program size

- Lines of Code
  - Ranging from 5 - 100
- Implemented from various online sources
  - “100+ python challenging programming exercises”
  - www.ActiveState.com -- code recipes
  - www.Cprogramming.com -- challenge
- Beginner level programs that contain common code elements
  - Simple numerical calculations: Factorial
  - Mathematical constants approximations: pi, e, sqrt(2)
  - Simple text input Calculator
  - etc.
Breaking and Fixing
Breaking and fixing, The breaking process

- Start with correct implementation
  - Used as an oracle to produce a test suite
- GI applied with reversed objectives.
  - Evaluated with unittest
- Evolution is stopped if a valid break is found.
- A program is broken if it:
  - Fails on at least 1 test case
  - Does not produce run time errors on at least half of the test suite
Breaking and fixing, The fixing process

- Objectives are:
  - Number of test cases passed
  - Size of edit list, i.e. number of changes to the broken program
- Runs for 50 generations (population of 20)
- Returns the overall best solution.
  - Fewest number of changes made to the program to pass the greatest number of test cases.
### Experiments, Line for line

<table>
<thead>
<tr>
<th>Program</th>
<th>Size LOC</th>
<th>Avg. size of breaker</th>
<th>Avg. evals -&gt; fixed</th>
<th>Avg. proportion of error variants</th>
<th>Avg. size of fixer</th>
</tr>
</thead>
<tbody>
<tr>
<td>count_digs_letters</td>
<td>9</td>
<td>1</td>
<td>15.2</td>
<td>75%</td>
<td>2.01</td>
</tr>
<tr>
<td>dict_square</td>
<td>5</td>
<td>1</td>
<td>6.3</td>
<td>68%</td>
<td>1.5</td>
</tr>
<tr>
<td>divisible_5</td>
<td>7</td>
<td>1</td>
<td>10.2</td>
<td>81%</td>
<td>3.7</td>
</tr>
<tr>
<td>even_digits</td>
<td>13</td>
<td>1</td>
<td>4</td>
<td>74%</td>
<td>1.2</td>
</tr>
<tr>
<td>factorial</td>
<td>5</td>
<td>N/A</td>
<td>N/A</td>
<td>100%</td>
<td>N/A</td>
</tr>
<tr>
<td>formula_this</td>
<td>8</td>
<td>1</td>
<td>6.2</td>
<td>72%</td>
<td>4.1</td>
</tr>
</tbody>
</table>
## Experiments, Line for line

<table>
<thead>
<tr>
<th>Program</th>
<th>Size LOC</th>
<th>Avg. size of breaker</th>
<th>Avg. evals -&gt; fixed</th>
<th>Avg. proportion of error variants</th>
<th>Avg. size of fixer</th>
</tr>
</thead>
<tbody>
<tr>
<td>lines_2_list</td>
<td>12</td>
<td>1</td>
<td>10.9</td>
<td>67%</td>
<td>4.01</td>
</tr>
<tr>
<td>list_tuple</td>
<td>5</td>
<td>N/A</td>
<td>N/A</td>
<td>100%</td>
<td>N/A</td>
</tr>
<tr>
<td>make_multiMatrix</td>
<td>8</td>
<td>1</td>
<td>14.5</td>
<td>80%</td>
<td>3.4</td>
</tr>
<tr>
<td>sort_unique</td>
<td>5</td>
<td>1</td>
<td>13.2</td>
<td>45%</td>
<td>2.13</td>
</tr>
<tr>
<td>sort_words</td>
<td>5</td>
<td>1</td>
<td>8.4</td>
<td>51%</td>
<td>1.25</td>
</tr>
</tbody>
</table>
Experiments, Summary of line for line

- **Breaking**
  - Fitness is effectively binary: broken or not broken
    - pass all or no test cases
  - Highly unlikely programming errors.
    - e.g. forgetting a complete line?
  - Takes only one line out of place to break.
  - If a valid break exists it is found in first generation.

- **Fixing**
  - Takes longer to find the fix than the break
  - High proportion of variants do not run
    - and those that run are mostly semantically identical, i.e. loads of redundancy
Experiments, finer grained

Case example, Dictionary of squares

- Input: single integer n
- Output: dictionary of all the numbers squared from 0 to n
- 5 test cases which include boundary inputs, n = 0 and 1
- Program was broken by replacing the first occurrence of 1 with 2.
  - <REPLACE, '1', '2', 2,15>
- Then the GI was run 100 times to fix.
  - No elitism

```python
def dict_squares(n):
    d=dict()
    for i in range(1,n+1):
        d[i]=i*i
    return d
```
Experiments, Finer grained: Dictionary of squares
Average fitness of fixes to dict_squares over 100 experiments

Experiments, finer grained: Dictionary of squares
Experiments, finer grained

Case example: A simple text input calculator

- ~100 LOC
- Inserted bugs with 4 edits
  - Forced by increasing the required failed test cases
  - `<REPLACE, '*', '+', 24, 4>`<REPLACE, '-', '+', 22, 4>`<REPLACE, '/', '**', 36, 4>`<REPLACE, '+', '%', 20, 4>`
- Fails all test cases (19)
  - At least one test case for each function: +, -, *, and /
  - and the rest combines them
- Again: GI run 100 times to fix
  - Now with elitism
Experiments, finer grained

Recorded fitness for a single run

Number of test cases passed (max 5)

Generation
**Experiments, summary of finer grained**

- Sometimes finds mutations that pass some test cases
  - Fitness is not always binary, rather a step: passes 1 or 2 boundary cases.
  - More bugs -> more needles
- Much more **realistic** programming errors
  - Typing “=” instead of “+=” or “<” instead of “<=”
- Only one edit needed to break
Experiments, summary of finer grained

- We can nearly always find a valid break
  - Syntactically correct programs
  - High proportion of variants run
- For such small programs the fix is usually converting it back to the original.
  - No clever fixes, that weren’t foreseen.
- The fix is most often found in the first 5-10 generations.
- Still, finding the fix takes much longer than finding the break.
  - In practice “Needle/s in a haystack” fitness function that is largely level.
Summary
Summary

- GI for Python programs is doable and promising
- Tested on multiple small programs
- Considered 2 dimensions of search granularity
  - Step size
  - Movable code
- Line based GI is not a realistic option for small programs
  - Where the boundary of size lies remains to be confirmed
- Smaller programs call for finer grained searches
Thanks for listening
Questions?