Word Count
as a Traditional Programming Benchmark Problem for Genetic Programming

Tom Helmuth and Lee Spector
Traditional Programming
Problems in GP

- Mimic human programming
- Large instruction set
  - multiple data types
  - control flow
  - I/O
- Based on tests
  - input/output example behavior
Traditional Programming Problems in GP

• Need benchmark problems!
  ○ interest shown in community survey\(^1\)
  ○ but, none recommended in survey paper

• Word count problem

Unix Command wc
Unix Command wc

newlines
words
characters
Why wc Makes An Interesting Traditional Programming Problem

- Requires multiple data types
- Imitates real program
- Difficult but reasonably fast
- Open source, easy to implement
- Generalization to unseen test cases
Generate wc Problem Instance:
Test Cases

- 0 to 100 character files
- Random string
  - 200 training set -- 500 test set
- Random string ending in newline
  - 20 training set -- 50 test set
- Edge cases
  - 22 training set
  - examples: "", "A", "\n", "\n" repeated for 100 chars
Example Experiment

- Compare parent selection techniques
  - lexicase selection
  - tournament selection
  - implicit fitness sharing selection
Lexicase Parent Selection

- Emphasizes individual test cases
  - not aggregated fitness across test cases
- Uses random ordering of test cases for each selection event
- Unlike in Pareto selection, some test cases provide more selection pressure than others
Lexicase – Pseudocode

To select single parent:
1. Shuffle test cases
2. First test case – keep best individuals
3. Repeat with next test case, etc.
   a. Until one individual remains
Push and PushGP

- **Push** - Stack-based language for GP
  - Arguments and results from typed stacks
  - Executing code also on stack

- **PushGP** - Mostly typical GP using Push

http://pushlanguage.org
Instructions

- **General purpose:**
  - I/O
  - control flow
  - tags for modularity
  - string, integer, and boolean
  - random constants

<table>
<thead>
<tr>
<th>Input</th>
<th>file_readchar, file_readline, file_EOF, file_begin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>output_charcount, output_wordcount, output_lincount</td>
</tr>
<tr>
<td>Exec</td>
<td>exec_pop, exec_swap, exec_rot, exec_dup, exec_yank, exec_yankdup, exec_shove, exec_eq, exec_stackdepth, exec_when, exec_if, exec_do<em>times, exec_do</em>count, exec_do*range, exec_y, exec_k, exec_s</td>
</tr>
<tr>
<td>Tag ERGs</td>
<td>tag_exec, tag_integer, tag_string, tagged</td>
</tr>
<tr>
<td>String</td>
<td>string_split, string_parse_to_chars, string_whitespace, string_contained, string_reverse, string_concat, string_take, string_pop, string_eq, string_stackdepth, string_rot, string_yank, string_swap, string_yankdup, string_flush, string_length, string_shove, string_dup</td>
</tr>
<tr>
<td>Integer</td>
<td>integer_add, integer_swap, integer_yank, integer_dup, integer_yankdup, integer_shove, integer_mult, integer_div, integer_max, integer_sub, integer_mod, integer_rot, integer_min, integer_inc, integer_dec</td>
</tr>
<tr>
<td>Boolean</td>
<td>boolean_swap, boolean_and, boolean_not, boolean_or, boolean_frominteger, boolean_stackdepth, boolean_dup</td>
</tr>
</tbody>
</table>
| ERC        | Integer from [-100, 100]  
|            | {"\n", "\t", "_"}  
|            | {x | x is a non-whitespace character} |
## PushGP Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runs Per Condition</td>
<td>200</td>
</tr>
<tr>
<td>Fitness Evaluations Budget</td>
<td>72,600,000</td>
</tr>
<tr>
<td>Population Size</td>
<td>1000</td>
</tr>
<tr>
<td>Max Generations</td>
<td>300</td>
</tr>
<tr>
<td>Max Program Size</td>
<td>1000</td>
</tr>
<tr>
<td>Max Initial Program Size</td>
<td>400</td>
</tr>
<tr>
<td>Max Node Evaluations</td>
<td>2000</td>
</tr>
<tr>
<td>Genetic Operator</td>
<td>ULTRA (100%)</td>
</tr>
<tr>
<td>ULTRA Mutation Rate</td>
<td>0.01</td>
</tr>
<tr>
<td>ULTRA Alternation Rate</td>
<td>0.01</td>
</tr>
<tr>
<td>ULTRA Alignment Deviation</td>
<td>10</td>
</tr>
</tbody>
</table>
Performance Metrics for Traditional Programming Problems

- When comparing sets of runs, don’t use mean best fitness
  - don’t care about incremental improvements of GP
- Care about perfect solutions
  - must pass training and unseen test sets
- Compare success rates
Success Rates

- Fisher’s exact test for significance
- Confidence intervals on difference
## Results

<table>
<thead>
<tr>
<th>Selection</th>
<th>Tournament Size</th>
<th>Successes (200 runs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lexicase</td>
<td>-</td>
<td>11</td>
</tr>
<tr>
<td>Tournament</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>Implicit Fitness</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Sharing</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>
Results

- 95% confidence interval: [0.020, 0.088]
- Small but meaningful differences
Conclusions

● More traditional programming in GP!
  ○ problems/benchmarks
  ○ wc problem good starting point
  ○ applications
● Lexicase selection

Acknowledgments: This material is based upon work supported by the National Science Foundation under Grants No. 1017817 and 1129139. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.