Word Count as a Traditional Programming Benchmark Problem for Genetic Programming

Thomas Helmuth
Computer Science
University of Massachusetts
Amherst, MA 01003
thelmuth@cs.umass.edu

Lee Spector
Cognitive Science
Hampshire College
Amherst, MA 01002
lspector@hampshire.edu

ABSTRACT

The Unix utility program wc, which stands for "word count," takes any number of files and prints the number of newlines, words, and characters in each of the files. We show that genetic programming can find programs that replicate the core functionality of the wc utility, and propose this problem as a "traditional programming" benchmark for genetic programming systems. This "wc problem" features key elements of programming tasks that often confront human programmers, including requirements for multiple data types, a large instruction set, control flow, and multiple outputs. Furthermore, it mimics the behavior of a real-world utility program, showing that genetic programming can automatically synthesize programs with general utility. We suggest statistical procedures that should be used to compare performances of different systems on traditional programming problems such as the wc problem, and present the results of a short experiment using the problem. Finally, we give a short analysis of evolved solution programs, showing how they make use of traditional programming concepts.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming—*Program synthesis*

Keywords

genetic programming; traditional programming; PushGP; benchmark; lexicase selection

1. INTRODUCTION

The Unix utility wc is a command that prints the newline, word, and character counts in one or more specified files[1]. For each given file, wc's first output counts the newlines in the file, which can be as low as zero if all characters are on the same line. Its second output gives the number of words in the file, defined as contiguous runs of non-whitespace characters separated by whitespace characters (spaces, tabs, and newlines). Its third output gives the number of characters in the file.

We would like to investigate automatic program synthesis techniques such as genetic programming (GP), in particular their ability to solve traditional programming problems such as replicating the the behavior of the wc utility. Automatically synthesizing a wc program presents a variety of challenges tackled routinely by software programmers but not handled well by most GP systems. In this paper we will present the wc problem as a challenging but solvable GP problem, for which programs that duplicate the wc utility can be synthesized without knowledge beyond a data set of desired behavior tests consisting of example input/output pairs.

The GP community has recently discussed the need for better benchmarks [5, 18]. These discussions provide recommended benchmarks for some problem domains, but not for traditional programming problems (which they call "algorithmic programming" problems), despite the traditional programming category receiving the second most votes in a community survey asking what types of problems a GP benchmark suite should contain [18]. Traditional programming problems mimic the programming of human programmers, requiring multiple data types, control flow, and a large instruction set. Automating human programming tasks has long been a goal of GP, as articulated in Koza's first book [4] for example. The absence of traditional programming problems among current recommended benchmarks is unfortunate and should be addressed.

The wc problem can be used as a traditional programming benchmark problem with many desirable qualities. Since this problem requires one to find a program that performs a task available as a Unix utility command, it shows that GP can be used to solve a useful problem. The wc problem is solvable yet non-trivial using GP, which we will show with an experiment. Relative to currently existing program synthesis technology, it strikes a good compromise between difficulty and the ability to perform many runs quickly; we were able to perform 200 GP runs per condition in a reasonable amount time using a small cluster. The problem is open source, representation independent, and easily implemented; we have done so in the PushGP system, which is available for free[2]. This paper provides a precise definition of the wc problem, including recommendations for computational resources, instruction sets, and training and test data. We

[1]See, e.g. `http://www.linuxmanpages.com/man1/wc.1.php`

[2]`https://github.com/lspector/Clojush`

also make recommendations for comparing the performance of different systems on this problem.

In the next Section we describe the PushGP genetic programming system, which we used to conduct the experiments described in Section 5. Following this we give a detailed description of the proposed wc benchmark problem, including the data set and instruction set we recommend. Section 4 discusses performance measures and statistical procedures appropriate for comparing GP systems on the wc problem. We then present our experimental results and conclusions.

2. PUSH AND PUSHGP

The experiments described in Section 5 use the PushGP genetic programming system, which evolves programs in the Push programming language. Push was designed specifically for use in GP systems as the language in which evolving programs are expressed [11, 17, 15]. Push is a stack-based programming language similar in some ways to others that have been used for GP (e.g. [8]). Although any general purpose GP system could be applied to the wc problem, Push provides many tools important for traditional programming problems, such as multiple types, control flow, and the ability to return multiple outputs. We chose to use PushGP in our experiments because of its ease of use for traditional programming problems; much of PushGP was designed with this type of problem in mind. The experiments we present in Section 5 are simply an example of using the wc benchmark; since they are not the focus of the paper, the choice of the GP system has little bearing on our analysis of the benchmark and will not be discussed further.

Push is a postfix, stack-based programming language. Interpreted Push programs push literals onto data stacks, and instructions act on stack data, returning their results to stacks. A separate stack is used for each data type; instructions take their arguments (if any) from stacks of the appropriate types and leave their results (if any) on stacks of the appropriate types. This use of multiple stacks allows instructions and literals to be freely intermixed regardless of type while still ensuring type safety. Instructions that find insufficient data on the relevant stacks act as "no-ops"—that is, they do nothing.

"Code" is itself a Push data type, and Push programs can easily (and often do) manipulate their own code as they run; this permits the expression of complex and novel control structures. The execution of a Push program is interpreted on the exec stack, and instructions that manipulate the exec stack can change the program itself as it runs. Various exec stack instructions allow for the concise expression of iteration, recursion, and conditional statements. Blocks of code can be hierarchically structured with parentheses, allowing exec-stack-manipulating instructions to affect larger chunks of code.

Push and PushGP implementations now exist in Clojure, Java, C++, JavaScript, Python, Ruby, Scheme, Common Lisp, and Scala. Many of these are available for free download from the Push project page[3].

3. WC PROBLEM

We wish to automatically create a program that replicates the functionality of the wc utility using a general-purpose program synthesis system that has not been tailored to this specific problem. We would like our program synthesis system to create such a program based entirely on examples of the desired input/output behavior. The evolved program should take a file as input and should output three integers representing the number of newlines, words, and characters in the file. An evolved solution should produce the correct results for any file between 0 and 100 characters in length; we chose 100 characters as a maximum length to ensure reasonable execution times when evaluating programs.

Our requirements for a successful program differ slightly from the full functionality of the wc utility command. In particular, the wc command prints the outputs, where we just require the program to use output instructions to return the results. Additionally, the wc command can take multiple file inputs and print the output for each, along with summed totals across the files. Of course, a human programmer could take an evolved solution to the wc problem and use it as the core functionality of a program to perfectly replicate the full wc command. This would only require a loop over the input files which calls the evolved program and formats the results. If a fully evolved wc command was desired, we could likely evolve the iteration and printing part of the utility separately, which would use the evolved newline, word, and character counting function (presented as the primary objective here) as a subroutine. A reasonable avenue of future work could use this extension to show how complete software can be evolved based on high-level descriptions of the modularity of the software and behavioral examples for each module.

A GP problem is defined primarily by the labeled data set it uses to evaluate evolved programs and the instructions used in random code generation. The following section defines the data set to be used with the wc problem, and Section 3.2 presents the instructions we recommend for the wc problem.

3.1 Data set

The goal of the wc problem is to find a program that correctly outputs the number of newlines, words, and characters in any given file containing 100 characters or fewer. We chose to limit the available characters to non-whitespace ASCII characters (i.e. those with decimal values between 33 and 126) along with newline, space, and tab. In order to guide GP toward a solution program, we require a set of labeled training data. Additionally, we would like a set of labeled test data that the GP algorithm never sees in order to test the generality of evolved solutions.

From a set of file inputs we can produce the labeled data by running an existing word count function on each file and storing integers representing the numbers of newlines, words, and characters in the file. In our GP runs, we produce three error values (newlines, words, characters) for each test case, each the absolute difference between the expected output and the output of the program. These three errors could be combined into a single error per test case, but we keep them separate since some parent selection mechanisms treat errors across test cases differently. Since we can create correct outputs given an input, our task is reduced to deciding which files to use as training and test inputs.

The number of files containing 100 or fewer of the allowed characters far outnumbers the number of files we can use as training and test data. We must therefore either provide

[3]`http://hampshire.edu/lspector/push.html`

Table 1: Data domains for the wc problem.

Type	Train	Test
Random string	200	500
Random string ending in newline	20	50
Edge cases (see Table 2)	22	0

a data set of prescribed files or a method of creating randomly generated files as data. We use a combined method, holding some data constant across every run while randomly generating other data.

In order to facilitate this creation of training and test data for the wc problem and other traditional programming problems, we designed a general system for automatic data generation based on data domains. A "data domain" D is a set of program inputs described by either a list of inputs or a random input generator function. The sets {`"hi"`, `"hello"`, `"howdy"`, `"hey"`} and $\{s | \text{“}zoo\text{”} \text{ is a substring of } s; \text{len}(s) <= 100]\}$ are examples of data domains, where the former is an enumerated set of four inputs and the latter is a large set containing all strings of length less than or equal to 100 that contain the substring `"zoo"`. Along with each data domain D, the user must provide the integers $train(D)$ and $test(D)$ that indicate the number of training and test cases respectively to generate from D.

To generate training and test data from a set of data domains $\{D_1, D_2, ..., D_n\}$, we simply take each domain and create the required number of cases. If the domain D_i is an enumerated list of inputs, we select $train(D_i)$ and $test(D_i)$ of them at random, without replacement within the training cases or test cases. If the domain is described by a random input generator, we run it $train(D_i)$ and $test(D_i)$ times (with replacement) to create the data. This automatic data generation system allows for the generation of training and test cases for a wide range of problems.

Table 1 summarizes the data domains we recommend for the wc problem. The first data domain, from which we draw the most training and test cases, creates random strings between 0 and 100 characters long. This random string generator first chooses a random size uniformly from the range $[0, 100]$ inclusive. It then creates a string of random characters, biasing each character selection to choose a space 35% of the time, newline 10% of the time, tab 5% of the time, and any other character 50% of the time. This bias, although somewhat arbitrary, seems to result in a good variety of whitespace and non-whitespace characters to cover a range of inputs. This random string generator has a higher percent of newlines and tabs compared to English text and program language code; still, it empirically does a good job of covering many interesting types of files.

The second data domain in Table 1 uses a random string generator identical to the first data domain, except that it ensures that the last character in the string is a newline. In our initial exploration of this problem, we did not always require some data cases that end in a newline. In those runs, we noticed that a number of evolved solution programs did not generalize to file inputs ending in newlines, so we added a relatively small set of them to ensure evolution sees this type of input.

Although the randomly generated strings created by the first two data domains usually cover a wide range of inputs, they often omit specific edge cases. Exploratory GP runs evaluated on random inputs often resulted in solutions that did not generalize, particularly to very small inputs and inputs with many newlines or words. Our third data domain in Table 1, expanded in Table 2, addresses this issue by ensuring that every run includes a set of training cases that covers these problem edge cases. The first 11 of these inputs represent short files containing limited characters. The last 11 edge cases are long strings that have 100 characters each, and include files with many more lines and words than most randomly generated inputs. The inclusion of these edge cases in the training data seems to improve our system's ability to evolve general solutions.

Table 2: This table contains the 22 edge case training cases used in our wc problem runs. Each case attempts to cover a possibly tricky input that may not be exemplified in the random string inputs. Here \n represents a newline, \t represents a tab, and ␣ represents a space.

1	`""`
2	`"A"`
3	`"␣"`
4	`"\t"`
5	`"\n"`
6	`"5"`
7	`"#"`
8	`"A\n"`
9	`"␣\n"`
10	`"\t\n"`
11	`"\n\n"`
12	`"\n"` repeated, 100 characters
13	`"␣"` repeated, 100 characters
14	`"\t"` repeated, 100 characters
15	`"A"` repeated, 100 characters
16	`"\nA"` repeated, 100 characters
17	`"\n\nB"` repeated, 100 characters
18	`"CD\n"` repeated, 100 characters
19	`"E␣"` repeated, 100 characters
20	`"F\t"` repeated, 100 characters
21	`"x\ny␣"` repeated, 100 characters
22	`"␣\n"` repeated, 100 characters

Overall, we recommend the use of 242 training cases and 550 withheld test cases. These numbers seem to strike a reasonable balance between providing enough training data to guide evolution toward a general solution while not requiring the evaluation of evolving programs on an unreasonable number of training cases.

Note that GP practitioners need not use every training case during every program evaluation; they may choose to use a subset of the training cases for some or all evaluations. Some recent GP techniques such as co-evolved fitness predictors [9, 10] and spatial co-evolution [2] require fewer training cases per evaluation. Others may wish to use fewer training cases in order to lessen the number of program evaluations required, possibly increasing the performance of GP compared to the number of evaluations. In any case, the training cases used for evaluation should be taken exclusively from the training data set, which should remain static throughout the GP run even if its subset used for evaluation does not. Also, practitioners using fewer evaluations per program must take care when reporting results, as we discuss in Section 4. Finally, no matter what subset of the training set is used for evaluating programs, the entire training and test sets should be used to verify the generalization of evolved solutions.

3.2 Instruction Set

Traditional programming problems require much larger and more diverse instruction sets than many other problems solved by GP. As such, we do not want to specify a particular instruction set that should be used for the wc problem, but instead give some guidelines as well as the instruction set we use in our experiments. This allows others to compare results using different representations and instruction sets.

Most typical instructions from programming languages, whether designed for GP or general purpose programming, are allowable. This list could include control flow instructions for looping or conditionals, functional instructions such as mapping and filtering, or string manipulation instructions. We encourage researchers working on the wc problem to use their best judgment when choosing instruction sets, noting only that one should obviously not include instructions that have been tailored specifically to this problem. To give an extreme example, instructions that return the newline count, word count, or character count in a file would immediately solve most of the problem and should obviously be avoided.

The wc problem, unlike many problems in GP, requires programs to read input from files. The input instructions should give the evolving programs enough power to solve the task without making it trivial. They should also be reasonably similar to input instructions in modern high-level programming languages. For example, providing a `file_readword` instruction that reads the file until the next whitespace character seems inappropriately powerful, especially considering such an instruction does not exist in most programming languages. We also elected to omit an instruction that reads the entire file, since it seems to provide too much power when combined with other string manipulation instructions, although an argument could be made for its inclusion.

Table 3 gives the Push instructions we include in our instruction set. The "Input" row gives the file input instructions. The `file_readchar` and `file_readline` instructions read a single character and a single line from the file respectively, keeping track of the location in the file. They both do nothing if the end of the file has been reached. These instructions are similar to those found in many higher-level languages. The `file_EOF` instruction pushes true onto the Boolean stack if the end of the file has been reached, and false otherwise. Finally, `file_begin` moves the file "cursor" to the beginning of the file. In our implementation, the input instructions actually just manipulate and read sections of a string instead of an actual file, but we could easily change those instructions so that an evolved program actually reads a file from the disk.

The wc problem requires programs to output three integers, one each for the character count, word count, and newline count. One approach, which we employ, is to use output instructions that store a return value. It is less clear what the best approach is for a fully functional language, whose programs traditionally only produce one value. Our three output instructions in the second row of Table 3 each take the top value on the integer stack and store it in a variable, which is later used as the output at the end of execution. If an output instruction is called multiple times during the execution of a program, each execution overwrites the previous, meaning that the value stored during the final execution is the one given as output.

Table 3: Push instructions included in our instruction set.

Input	`file_readchar, file_readline, file_EOF, file_begin`
Output	`output_charcount, output_wordcount, output_linecount`
Exec	`exec_pop, exec_swap, exec_rot, exec_dup, exec_yank, exec_yankdup, exec_shove, exec_eq, exec_stackdepth, exec_when, exec_if, exec_do*times, exec_do*count, exec_do*range, exec_y, exec_k, exec_s`
Tag ERCs	`tag_exec, tag_integer, tag_string, tagged`
String	`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`
Integer	`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`
Boolean	`boolean_swap, boolean_and, boolean_not, boolean_or, boolean_frominteger, boolean_stackdepth, boolean_dup`
ERC	Integer from [-100, 100] {`"\n"`, `"\t"`, `"␣"` } $\{x \mid x$ is a non-whitespace character$\}$

The exec instructions given in the third row of Table 3 provide the ability to define and execute arbitrary control structures within a Push program, and make the language Turing-complete. Each of these instructions manipulates the top of the exec stack, thus changing what instruction will be executed next. The first nine exec instructions simply allow for normal stack manipulations. `exec_when` and `exec_if` provide conditional execution of the following parenthesized block(s) of code. The next three instructions provide explicit looping. Finally, the last three instructions are combinators that allow for recursion (see [15]).

Tagging provides for another method for manipulating control flow in Push [16]. The `tag_exec`, `tag_integer`, and `tag_string` ERCs (ephemeral random constants) create instructions, each of which has a name that specifies an associated integer tag. For example, the instruction `tag_string_392` specifies the tag 392. Each instruction created by one of these ERCs takes the top item from the indicated stack and tags it with the associated tag. For example, `tag_string_392` takes a string from the string stack and tags it with 392. The `tagged` ERC creates instructions that retrieve tagged items by closest tag match[4] and push the retrieved items onto the exec stack. For example, after tagging a string with 392, the instruction `tagged_344` would push the tagged string onto the exec stack, assuming no other data had been tagged with a tag between 344 and 392. Since tagging instructions can tag code from the exec stack, tags can be used to create subroutines that become available to the rest of the program. The inexact matching used in tag-based retrieval allows the number of tags used in programs to grow incrementally over evolutionary time [13].

The wc problem obviously requires significant string manipulation to count newlines, words, and characters in files. The string stack instructions in Table 3 provide diverse string manipulations without making the problem trivial. The instruction `string_split` takes a string argument and splits it on whitespace characters, returning the results to the string stack. `string_parse_to_chars` takes a string and pushes each individual character string onto the string stack. `string_whitespace` returns the boolean true if the string argument consists entirely of whitespace characters, and false otherwise. The remaining string instructions consist of standard string manipulation instructions and stack instructions.

The integer and boolean instructions we provide in our instruction set are all normal stack manipulation instructions or integer/boolean operators. Integer instructions are necessary, since the outputs for the problem are integers from the integer stack. The exec stack instructions that allow for conditional expressions rely on the boolean stack for their arguments.

We provide three ERCs that create random constants during random code generation. The first creates integers in the range $[-100, 100]$. The second randomly selects a single whitespace character string: newline, tab, or space. The third creates a string with a single non-whitespace character.

4. WC PERFORMANCE MEASURES

The primary goal of a benchmark problem is to compare the characteristics, in particular performance, of different methods or algorithms on problems with similar traits to actual applications of those methods. As such, we must determine how to measure the performance of GP systems on the wc problem.

The most frequently used performance metrics in GP papers include success rate, computational effort, and mean best fitness. Success rate measures the percent of runs performed that find a successful program (sometimes called an "ideal solution"). Computational effort extends success rate to estimate the number of program evaluations necessary to find a successful program with high probability [4]. The best fitness of a run is simply the best fitness achieved by a program during the run; the mean best fitness of a set of runs is the mean of the best fitnesses found in the runs.

Recent discussions of benchmarks in GP have criticized the use of success rate and computational effort, instead recommending either best fitness of run or testing with a withheld generalization data set [5, 18]. The main criticisms cited are that success rate and computational effort "measure how well a method solves trivial problems" and that they have "poor accuracy and statistical invalidity" [5]. Although we agree that computational effort may be statistically problematic, we believe that for the wc problem and many other traditional programming problems success rate remains the best method for measuring performance, especially when used in conjunction with a withheld test set.

The goal of the wc problem is to find a program that *perfectly* counts the numbers of newlines, words, and characters in a file. If a GP system finds a program that returns the correct outputs except for when the input file contains the character `Q`, it is of little importance even if it otherwise achieves good fitness. In essence, no one cares about incremental improvements in discovered programs unless one is found that perfectly passes all of the training cases. Therefore, when measuring performance of a GP system on the wc problem, best fitness contributes little useful information.

The success rate of a set of GP runs gives more information than best fitness about how well the system performs on the wc problem, since it measures how often the system finds perfect programs. We do agree that programs that achieve perfection on the training data should be tested on withheld generalization data, which we describe for the wc problem in Section 3.1. A program should only count as a successful program if it achieves zero error on both the training data and the test data. The success rates we report only include programs that have perfect error on the training and test data, although we also note the numbers of programs that achieve perfect error on the training data only.

We argue more generally that the ability to solve a traditional programming benchmark problem perfectly does not indicate the triviality of the problem. If our goal is to someday have GP evolve complex software that performs important functionality without having to be coded by humans, we would hope that such a system would pass all of its tests before going into use. The purpose of traditional programming benchmark problems is to compare the performances of GP methods on traditional programming problems; as such, we would expect GP to be able to solve them in order to have a hope of solving real programming problems. Although the wc problem is simpler than almost all software systems of real value, it is useful enough to be included as a Unix utility, and offers more interesting requirements than other suggested traditional programming problems such as list sorting.

4.1 Computational Budget

When comparing success rates of different GP methods, one must be careful to ensure each method receives similar amounts of computation. Using CPU time or wall time as a proxy for computational requirements has many flaws, including differing greatly for the same computation based on the machine used, the machine's load, and the optimization of the GP system. Instead, since program evaluations often dominate the runtime of GP algorithms, most comparisons in the literature use a unit related to program evaluation to measure the computation used by the system. The computational effort measurement tries to take this into account by taking the population size and generation of found solutions into account, but we would like to avoid using computational effort. Another approach prescribes a budget of some sort to limit the computation of the GP system. Options for computation budgets offered by the community survey in [18] include, in increasing level of granularity: number of

[4]The closest match for a reference to tag t_{ref}, among all tags t_{val} that have been used to tag values, is the tag t_{val} for which $t_{val} - t_{ref}$ is minimal if at least some $t_{val} \geq t_{ref}$, or maximal if all $t_{val} < t_{ref}$ [13].

generations, number of program fitness evaluations, number of fitness case evaluations (i.e. number of times any program is executed), and number of node evaluations.

Using the number of generations as the computational budget is equivalent to using the number of program fitness evaluations if the population size is also prescribed, except that it does not allow for testing varying population sizes. The number of program fitness evaluations provides a good level of control, assuming that each GP system tests every fitness case during every program evaluation. But, as mentioned in Section 3.1, some recent GP techniques require a small subset of the fitness cases be used during each program evaluation [9, 2]. These techniques would be at a disadvantage if they must adhere to numbers of program evaluations equivalent to techniques that use every fitness case during evaluation. In fact, both of these papers report results relative to the number of fitness case evaluations in order to make the comparisons fair. The number of node evaluations seems too fine of a measurement that could vary largely based on the GP system and how high-level the language is in which programs evolve. This method may be more accurate than using number of fitness case evaluations when using a single GP system with the same instruction set, but likely includes too much variation when comparing different GP systems.

Of these options, we recommend using a maximum budget based on the number of fitness case evaluations to limit the computation of the compared GP methods. This method allows for flexibility in many areas of algorithm design while ensuring systems receive similar computation. For the wc problem, we use the 242 training cases given in Section 3.1 for every program evaluation. We use a maximum of 300 generations with a population size of 1000, giving us at most 72,600,000 fitness case evaluations per run. To produce results that could be compared against those provided here, we recommend using a similar limit.

4.2 Statistical Procedure

In order to determine the statistical significance and reliability of the difference in success rates found by different GP methods, we recommend the use of both Fisher's exact test and confidence intervals of the difference in success rates.

Fisher's exact test tests the null hypothesis that there is no association between two compared methods and their number of successes. This test returns a p-value, which can indicate a rejection of the null hypothesis if it is below a prespecified significance level, usually 0.05. This test behaves similarly to the chi-squared test, except that it is much more accurate when the number of successes or failures for either of the compared methods is near zero; if the number of successes and failures for each method is sufficiently large, the chi-squared test is also acceptable.

There has recently been increasing criticism of null hypothesis significance testing in the sciences, for example [1], which instead recommends the use of confidence intervals to indicate the reliability and precision of results. To supplement Fisher's exact test, we recommend reporting the difference in success rates along with a confidence interval of the difference. If each method produces at least 10 successes and failures, the large-sample confidence interval for comparing two proportions can be used. Otherwise, Moore and McCabe recommend the use of the plus four confidence interval for comparing two proportions, by adding one success and one failure to the observed results for each method [7]. This correction is effective as long as there are at least 5 runs from each method.

Table 4: PushGP parameters used in our experiment.

Runs Per Condition	200
Fitness Evaluations Budget	72,600,000
Population Size	1000
Max Generations	300
Max Program Size	1000
Max Initial Program Size	400
Max Node Evaluations	2000
Genetic Operator	ULTRA (100%)
ULTRA Mutation Rate	0.01
ULTRA Alternation Rate	0.01
ULTRA Alignment Deviation	10

5. EXPERIMENTAL RESULTS

We would like to show that the wc problem can be solved within the constraints we've recommended, without being too easy. We designed a simple experiment using PushGP comparing two parent selection methods, primarily to show the utility of the wc problem as a benchmark. The source code for our experiment is available freely online[5]. In this experiment, we will compare traditional tournament parent selection, with tournament size 7, to a recent selection algorithm called lexicase selection [12].

When lexicase selection is used, parent selection is not based on the total error of individuals aggregated over fitness cases, as it is in tournament selection and most other traditional parent selection methods, but rather case-by-case errors considered in random order. To select a parent, the lexicase selection procedure first randomly shuffles the fitness cases and creates a pool of candidate parents that initially includes the entire population. It then removes from the pool any individuals that do not have the best error in the pool on the first fitness case in the random ordering. Assuming more than one candidate for selection survives, this process is iterated with the second fitness case, and so on, until either a single individual remains or all fitness cases have been used (in which case a random remaining candidate is selected).

Lexicase selection tends to select individuals that are very good on a subset of the fitness cases, even if they perform poorly on other fitness cases. Recent results suggest that this selection pressure allows GP to find perfect solutions to difficult problems more often than tournament selection [3], suggesting an experiment that investigates its performance on the wc problem.

Table 4 presents the PushGP parameters used in our experiment. We allow a maximum of 72,600,000 fitness evaluations, as we recommend when using the wc problem as a benchmark. In all runs we used ULTRA ("Uniform Linear Transformation with Repair and Alternation") as the only genetic operator [14]. ULTRA takes two parents and creates one child by applying uniform crossover and mutation operations. In this study we use a variation of ULTRA that does not allow mutation to add or remove parentheses, al-

[5]`http://thelmuth.github.io/GECCO_2014_WC_Materials/`

Table 5: This 2 x 2 table presents the numbers of successes and failures for 200 PushGP runs using either tournament selection with size 7 tournaments or lexicase selection. A program is counted as a success if it achieves zero error on the training data as well as the withheld test data.

Selection Method	Successes	Failures	Success Rate
Tournament	0	200	0
Lexicase	11	189	0.055

though parentheses may nonetheless be added or removed via ULTRA's crossover and repair steps.

Table 5 presents the results from our runs comparing tournament selection and lexicase selection. The runs using tournament selection found no programs that achieved perfect error on the training data. Runs using lexicase selection found 17 programs that achieved perfect error on the training data, 11 of which were also perfect on the withheld test data. Based on this data, Fisher's exact test gives a p-value of 0.001, so we can reject the null hypothesis at the 0.05 significance level that there is no association between the selection method used and the number of successes. The difference in success rate between these methods is 0.055, with a 95% confidence interval of $[0.020, 0.088]$. This confidence interval uses the plus four estimate of the difference in proportions, since, as described in Section 4.2, the large-sample confidence interval does not apply because tournament selection resulted in zero successes.

For our experiment, Fisher's exact test tells us that there is likely an association between the selection method and the number of successes in our runs. The confidence interval for the difference in success rate between lexicase selection and tournament selection shows it is likely that the difference is small but meaningful. We interpret these results to mean that lexicase selection improves PushGP's ability to find solutions to the wc problem over tournament selection. We further find it important that PushGP found zero solutions when using tournament selection, showing that the system seems to need selection pressure different from what tournament selection provides to solve the wc problem, at least given our other parameters. This also shows that the wc problem is challenging, making it more interesting that PushGP was able to solve it using lexicase selection

Why can lexicase selection find solutions while tournament selection fails? We believe that the main difference lies in the ways in which these selection algorithms focus attention on certain characteristics of errors across fitness cases. Tournament selection aims to select individuals with the lowest total error across all fitness cases, and will therefore rarely select an individual with the lowest error in the population on many fitness cases but a few very bad errors. Instead, it will often select individuals with many mediocre-to-good errors. For problems like wc where the goal is an individual with zero error, we think that often an individual that does really well on many cases but poorly on others is in some sense closer in the evolutionary space to a perfect solution than an individual with better total error but not as many elite fitness cases. Lexicase selection may be taking advantage of this effect by selecting individuals that are very good on the fitness cases that randomly appear at the start of the shuffled fitness case list. It appears that this selection pressure helps GP to find solutions on the wc problem, and it seems reasonable to hypothesize that it will also help to solve other traditional programming problems in which it is necessary to perform perfectly on every fitness case. The selection pressure given by lexicase selection is in some ways related to that given by implicit fitness sharing [6], though unlike implicit fitness sharing it gives advantages to individuals that are good at combinations of cases rather than single cases; this suggests a comparison of the two as a future avenue of study.

We also analyzed the 11 evolved successful programs to see if they make good use of traditional programming techniques. We first simplified each program to remove instructions that had no effect on its outputs, and then counted the occurrences of types of instructions remaining in the simplified programs[6]. We found moderate uses of iteration and recursion in some of the programs, although others seemed to use brute force looping by repeating the same instruction many times sequentially. Conditional statements were also used occasionally, although not as frequently as we had expected. Tagging instructions occurred very infrequently, meaning that our programs did not use tags to implement subroutine-like modularity. Every program made use of strings, integers, and booleans; some programs also used string and integer literals, with `"\n"` appearing most often. Every solution used multiple outputs, as required by the problem definition. Overall evolution made decent use of PushGP's available traditional programming techniques, but success rates may be raised by improving GP's ability to evolve solutions that use iteration, conditional constructs, and modular subroutines.

6. CONCLUSIONS AND FUTURE WORK

This paper introduces wc as a traditional programming benchmark problem for genetic programming systems. Our demonstrations show that genetic programming can indeed solve this problem, and that it provides a sufficiently difficult and useful benchmark in the context of current technology. We also give guidance to researchers who wish to use wc as a benchmark in future work, with respect to both experimental design (e.g. instruction sets) and performance analysis.

We conducted experiments using PushGP comparing two parent selection algorithms, tournament selection and lexicase selection, in order to assess the relative strengths of these algorithms on the task of automatically synthesizing wc programs. The results clearly favored lexicase selection, which is itself an interesting finding, but the larger point is that the wc problem is a suitable traditional programming problem for conducting such experiments.

The most obvious direction for future work is to attempt to solve the wc problem, as described here, with other genetic programming systems using comparable computational resources. Analysis of the relative performance of different genetic programming systems on this problem may help us to advance the state of the art for applying genetic programming to traditional programming problems.

Another direction for future work is to consider other common UNIX utilities as possible traditional programming problem targets for genetic programming. Many such util-

[6]The simplified Push source code of the 11 successful evolved programs is available online: `http://thelmuth.github.io/GECCO_2014_WC_Materials/`

ities will probably be beyond the reach of current genetic programming techniques, but work on such problems may bring us closer to achieving the long-term goal of using genetic programming to automate programming tasks usually undertaken by human programmers. Additionally, the full wc command, including the handling of multiple input files and the printing of results, could be evolved in pieces as described in Section 3. This method of human-guided evolution of software seems like an avenue for evolving entire complex software systems.

The wc problem provides a starting place for traditional programing benchmarks for genetic programming. We would like to see the inclusion of the wc problem and other traditional programming problems in recommended genetic programming benchmark suites. Benchmark problems provide more information when techniques are compared on multiple problems. Such benchmark suites may be composed of problems from different classes in order to test general genetic programming systems, or entirely composed of traditional programming problems to test systems more specifically aimed at traditional program synthesis.

7. ACKNOWLEDGMENTS

Thanks to the members of the Hampshire College Computational Intelligence Lab, Yuriy Brun, and David Jensen for discussions that helped to improve the work described in this paper, to Josiah Erikson for systems support, and to Hampshire College for support for the Hampshire College Institute for Computational Intelligence. 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.

8. REFERENCES

[1] G. Cumming. The new statistics: Why and how. *Psychological Science*, 25(1):7–29, 2014.

[2] R. Harper. Spatial co-evolution: quicker, fitter and less bloated. In *GECCO '12: Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*, pages 759–766, Philadelphia, Pennsylvania, USA, 7-11 July 2012. ACM.

[3] T. Helmuth and L. Spector. Evolving a digital multiplier with the pushgp genetic programming system. In *GECCO '13 Companion: Proceeding of the fifteenth annual conference companion on Genetic and evolutionary computation conference companion*, pages 1627–1634, Amsterdam, The Netherlands, 6-10 July 2013. ACM.

[4] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.

[5] J. McDermott, D. R. White, S. Luke, L. Manzoni, M. Castelli, L. Vanneschi, W. Jaskowski, K. Krawiec, R. Harper, K. De Jong, and U.-M. O'Reilly. Genetic programming needs better benchmarks. In *GECCO '12: Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*, pages 791–798, Philadelphia, Pennsylvania, USA, 7-11 July 2012. ACM.

[6] R. I. B. McKay. Fitness sharing in genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, pages 435–442, Las Vegas, Nevada, USA, 10-12 July 2000. Morgan Kaufmann.

[7] D. Moore, G. McCabe, and B. Craig. *Introduction to the Practice of Statistics.* W.H. Freeman, 2009.

[8] T. Perkis. Stack-based genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 148–153, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[9] M. D. Schmidt and H. Lipson. Co-evolving fitness predictors for accelerating and reducing evaluations. In *Genetic Programming Theory and Practice IV*, volume 5 of *Genetic and Evolutionary Computation*, pages 113–130. Springer, Ann Arbor, 11-13 May 2006.

[10] M. D. Schmidt and H. Lipson. Coevolution of fitness predictors. *IEEE Transactions on Evolutionary Computation*, 12(6):736–749, Dec. 2008.

[11] L. Spector. Autoconstructive evolution: Push, pushGP, and pushpop. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 137–146, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.

[12] L. Spector. Assessment of problem modality by differential performance of lexicase selection in genetic programming: a preliminary report. In *1st workshop on Understanding Problems (GECCO-UP)*, pages 401–408, Philadelphia, Pennsylvania, USA, 7-11 July 2012. ACM.

[13] L. Spector, K. Harrington, B. Martin, and T. Helmuth. What's in an evolved name? the evolution of modularity via tag-based reference. In *Genetic Programming Theory and Practice IX*, Genetic and Evolutionary Computation, pages 1–16. Springer, Ann Arbor, USA, 12-14 May 2011.

[14] L. Spector and T. Helmuth. Uniform linear transformation with repair and alternation in genetic programming. In *Genetic Programming Theory and Practice XI*, Genetic and Evolutionary Computation. Springer, Ann Arbor, USA, 2013.

[15] L. Spector, J. Klein, and M. Keijzer. The push3 execution stack and the evolution of control. In *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1689–1696, Washington DC, USA, 2005. ACM Press.

[16] L. Spector, B. Martin, K. Harrington, and T. Helmuth. Tag-based modules in genetic programming. In *GECCO '11: Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 1419–1426, Dublin, Ireland, 12-16 July 2011. ACM.

[17] L. Spector and A. Robinson. Genetic programming and autoconstructive evolution with the push programming language. *Genetic Programming and Evolvable Machines*, 3(1):7–40, Mar. 2002.

[18] D. R. White, J. Mcdermott, M. Castelli, L. Manzoni, B. W. Goldman, G. Kronberger, W. Jaśkowski, U.-M. O'Reilly, and S. Luke. Better GP benchmarks: community survey results and proposals. *Genetic Programming and Evolvable Machines*, 14(1):3–29, Mar. 2013.