Parameter Sweeps For Exploring GP Parameters

Michael E. Samples, Jason M. Daida, Matt Byom, Matt Pizzimenti

Center for the Study of Complex Systems and the Department of Atmospheric, Oceanic, and Space Sciences

The University of Michigan, 2455 Hayward Avenue

Ann Arbor, MI 48109-2143 USA

{msamples, daida}@umich.edu

ABSTRACT

This paper describes our procedure and a software application for conducting large parameter sweep experiments in genetic and evolutionary computation research. Both procedure and software allows a researcher to examine multivariate nonlinearities that are common in genetic and evolutionary computation. Experiments of this nature are well suited to distributed computing environments (such as Grids and clusters) and we present an automated system for conducting parameter sweep experiments on heterogeneous networks. Emphasis is placed on experimental sampling, distributed robustness, and data analysis. The parameter sweep experimental procedure is easily applicable to any experiment involving computer simulations but is particularly well suited for evolutionary computation experiments.

Categories and Subject Descriptors

I.6.7—Simulation Support Systems Environments; I.2.2—Automatic Programming Program Synthesis

General Terms

Experimentation, Algorithms, Performance

Keywords

Parameter Sweep, Experiment Management, Evolutionary Computation, Distributed Computation, Data Reduction.

1. INTRODUCTION

In recent years, a number of people have provided critiques of the existing genetic programming (GP) experimental methodology [9,10,15,17,19]. This paper continues that trend by analyzing the design and execution of GP experiments. Although GP is used in specific examples, the methodology presented in this paper is applicable to the wider field of genetic and evolutionary computation. When invoking a GP engine to solve a particular problem, the engine can be thought of as performing a search on the sample space of all programs expressible in a given language of functions and terminals. Results obtained from this search would be the products of many low-level nonlinear interactions. Rules which govern these low-level interactions and thus restrict outcomes would be highly dependent on engine configurations, which, when changed only slightly, could produce dramatically different outcomes. In mathematical notation, results R are produced by the function GP when applied to a problem p and a certain set of configurations:

$$R = GP\left(p, c_1, c_2, c_3, c_4, \cdots\right)$$

It is an experimentalist's goal to understand how R is yielded by specific configurations, but it is impossible to exhaustively test an infinite number of configurations. While the use of small experiments can lead to knowledge about a given configuration's behavior, the results cannot necessarily be extrapolated to other configurations due to GP system nonlinearities. As a result, general knowledge of the mapping's behavior must be constructed by testing multiple classes of configurations. If experiments are run under multiple configurations, one would be able to develop laws and theories that can encapsulate knowledge of the system being studied to increase our predictive abilities on configurations not yet studied. *Succinctly, this means that it is critical to test for an observed phenomenon by conducting multiple tests and changing input parameters. Experimentalists should be prepared to do this*. We call an experiment involving the sampling of multiple configurations of parameters a *parameter sweep experiment* (PSE).

This type of experimental methodology is not limited to GP and has often been practiced in other fields involving computer simulations. A diverse body of works – including those dealing with high-energy physics [3], biomedical molecular modeling [6], analog circuit analysis in electrical engineering [20], and agent-based modeling [2] – have all benefited from large parameter sweep experiments. While this type of methodology is not new to science, it is rarely practiced in our field. Some notable exceptions to this trend are found in Daida et *al*. [11], Daida, Samples et *al*. [12], Luke & Panait [15], and Luke & Spector [16]. Why is the number of large multi-configuration studies relatively small? In this paper we suggest that parameter sweeps are often prohibitively difficult for researchers and present an automated system **Commander**, which is designed as an aid in conducting large *generic parameter sweep experiments* in Grid computing environments. ***Commander is not a GP system, but rather a software utility that performs automated experiment management and data reduction for parameter sweep experiments.***

In Section 2, we discuss the difficulties of performing large software experiments, the role of previous software solutions, and the niche for Commander. Section 3 describes Commander's internal design and distributed architecture. Section 4 uses common GP case studies to show why parameter sweeps are a useful type of scientific exploration. Section 5 discusses the results and the future of parameter sweeps in scientific computing. Section 6 concludes.

GECCO'05, June 25-29, 2005, Washington D.C., USA.

2. PARAMETER SWEEPS ARE HARD

We begin with a few definitions. A *trial* in GP research is defined to be the execution of a GP engine with a specified set of parameters and an initial random number generator (RNG) seed. A *parameter* is broadly defined to be a modifiable setting that can influence some aspect of a GP engine's behavior, thus influencing a trial's results. Parameters can be numeric in nature (e.g., population size = 2048, maximum allowed generations = 51), or algorithmic (e.g., tournament selection, steady-state replacement). A *datapoint* is defined to be a collection of some number of trials with identical parameter settings but different initial RNG seeds. A *parameter sweep* is then defined to be a type of experiment in which multiple datapoints are examined by conducting a number of trials with different parameter values. Parameter sweeps can be one-dimensional (e.g., studying multiple datapoints with different population sizes) or multidimensional (e.g., studying a set of datapoints in which each datapoint has a different 2-tuple of population size and maximum allowable generations).

There are difficulties inherent to the experimentation processes that make large *n*-dimensional parameter sweep experiments more difficult to complete than small single-datapoint experiments:

- *Experimental Setup* – For small experiments involving a single datapoint it is acceptable to write a shell script capable of executing the associated binary file a number of times. Since configurations do not change between trials, parameters can be statically determined inside the script and precompiled inside the binary. In PSEs, any script that runs an experiment either needs to send a list of current parameters to the binary via command line or run a unique binary for every configuration of parameters to be tested. Satisfactory scripts are generally both lengthy and non-portable—meaning that the construction of a new experiment would involve large time-consuming changes to the scripts.
- *Required Computation Time* – A single trial in GP can be very CPU-intensive. Also, since GP is inherently nondeterministic it is necessary for a large number of trials to be executed for each configuration. Parameter sweeps amplify this effect since they would be, by design, evaluating a larger number of configurations. This effect can be somewhat ameliorated through the use of parallel processing.
- *Data Reduction* – Single datapoint experiments generally produce the same type of data inasmuch as they were generated by the same configuration. As a result, the extraction of relevant data is a straightforward process. In parameter sweep experiments, the extraction of relevant data often necessitates the use of scripts in arranging the data in meaningful ways. Also, when sweeping across multiple variables there are problems in tagging data sets with parameter configurations. In a similar fashion to the setup scripts, these analysis scripts can be difficult to maintain between large experiments as the configuration spaces change.

For several years, our own GP research group maintained a large collection of Perl scripts to partially automate the process of running parameter sweep experiments. To reduce computation time, we would partition an experiment among several machines, running several similar scripts on each machine. Each script would be manually executed on idle cluster machines in a multi-user environment. From these experiences, we learned that distributing our computational load has its own unique challenges:

- *Distribution Algorithms* – When running distributed processes on multiple machines, it was essential to have some way of automatically assigning tasks to those machines. Inefficient algorithms wasted valuable resources, but efficient algorithms were difficult to generate. The growth in recent years of Grid computing has suggested that distributed algorithms should take advantage of Grid protocols. This would introduce an additional difficulty; different Grids can operate under different job-control protocols.
- *Robustness in Job Management* – While remotely executing a job, there were a number of ways in which a node or process could fail. This means that a distributed trial may not complete unless our experimental method gracefully handled such failures. This often involved the continuous re-execution of trials until successful completion. Such a procedure demanded complex scripts with conditional behavior and methods for determining the validity of experimental data. Since the distribution and validation processes were written for a specific experiment, these scripts resulted in significant retooling times between experiments.
- *Data Collection* – Any data that was generated on one computer was typically stored on that machine. When operating on a group of computers, however, we needed to collocate the data to conduct analysis. Data transmission (for our group typically amounting to several hundred megabytes per computer) was susceptible to errors, so the collation process needed to be robust.

Our group has experienced all of these difficulties associated with the remote distribution of GP parameter sweep experiments. We would stress that the time-consuming aspect of experiment administration cannot be underestimated. These difficulties, which are inherent to parameter sweeps and distributed computation, led us to explore automated experiment administration.

We are not the first group to attempt to solve these problems. Five available software packages are worth noting: APST [6,7], Condor [3,22], Drone [4], Nimrod/G [1], and BOINC [5]. In general, these projects are designed to distribute a list of computational tasks across a network to remote computers. A downside of these packages is that only some can do PSEs. For example, Drone, and Nimrod/G are the only ones designed to support parameter sweeps, whereas APST, Condor, and BOINC require the presence of custom external scripts to control parameter sweep experiments. Another downside of previous works is their occasional lack of transparency and robustness on various Grid and cluster networks. APST, Condor, and Nimrod/G contain support for only a limited number of Grid protocols. Drone does not support common Grid protocols, but is designed for cluster environments where a user has remote access. It has also been out of development for four years and has problems with robustness in remote computation. BOINC takes a different approach, using large numbers of common desktop computers and distributing jobs through downloadable clients, such as SETI@home.

Comparing the advantages of other available solutions and our needs in GP for conducting parameter sweep experiments, we

have developed the following four criteria for a generic distributed parameter sweep engine:

- Commander should facilitate the *easy setup* of new parameter sweep experiments, including experiments involving previously created programs. This process should not be script-governed but should instead derive from a simple grammar capable of fitting any program we might want to use in a distributed fashion. There should be no reason that the program we are executing – such as a GP engine – needs to be rewritten or extensively modified before it can be used with Commander. Of the previous work, only Drone and Nimrod/G met these standards for easy experimental setup.
- Commander should support *automated data reduction and analysis*. When experiments finish, Commander should be able to create sets of graphs and charts that were requested by the user in advance. Previously, only Nimrod/G attempted this feature. In an ideal engine, the graphs and charts should operate on the parameters over which the experiment was conducted, allowing users to quickly see results from a parameter sweep. This could lead to low turn-around time between experiments. This feature would be particularly valuable for GP experimentalists in both problem feedback and final analysis. These standard graphical tools could encapsulate overwhelming amounts of information immediately available after a GP experiment, giving the user invaluable feedback about the correctness of the results.
- The system should *operate transparently and seamlessly* as middleware across many types of computing networks. It should be able to take advantage of large-scale Grid networks, but must also be able to take advantage of local clusters and desktop machines. In taking advantage of these remote resources, users should not need to specify or submit jobs in different manners – all remote distribution should happen transparently to the researcher. This system should be robust; data integrity should be verified at multiple steps.
- Commander should be a generic tool for scientific exploration of computational problems. It is presumed that users of this tool have little interests in writing setup algorithms, distribution algorithms for multiple Grid protocols, and large analysis scripts. As such, Commander should hide the operational decisions from the user, allowing the user to focus on specifying the type of experiment. This allows the researcher to spend time and effort on valuable work instead of tedious experiment administration.

Although Commander is not novel in either its ability to perform large parameter sweep experiments or its ability to run processes on remote networks, its blend of automated parameter sweep experiments with transparent, robust, distributed computation is unique. In the next section, we discuss the implementation-level details of Commander – operational design and architecture.

3. COMMANDER IMPLEMENTATION

Commander was developed entirely in Python —a platform-independent interpreted language— to aid in transparency and robustness. Unlike the previous solutions listed in Section 1.1, Commander relies on no prescribed Grid technology for distributing processes. Instead, it operates with a *host* and numerous *clients*, relying on any one of numerous Grid protocols or cluster scripts merely to remotely launch the client process. In the following sections, we refer to the name "Commander" when referring to the entire project —a collection of hosts and clients— but we refer to host and clients separately as such. Hosts maintain complete knowledge of the current experiments, while clients actually run jobs and generate results data. In this aspect, Commander uses a master-worker architecture, which can be shown to be automatically load-balancing.

While Commander is not dependent on any specific Grid technology, it requires a Python interpreter, a locally accessible Commander client, and a Subversion client. Subversion [8] is an open source version control system similar to CVS and is used to distribute experiment-specific information to the clients. Since all of these technologies are platform independent, Commander clients are also platform independent. We regularly run clients on Unix, Linux, and OSX platforms, spread across different Grid architectures, cluster networks, and desktop computers.

In Section 2, we described three areas in which an ideal parameter sweep engine would excel: experiment construction, support for distributed computing, and data validation and analysis. In describing the design, it is natural to break up the descriptions into these portions as well. Figure 1 is Commander's architectural diagram that services Sections 3.1, 3.2, and 3.3.

3.1 Experiment Construction

When running experiments, we assume that users possess platform-independent copies of all materials required to run the experiment. For example, users possess source code or byte code instead of merely precompiled platform-specific binaries. This should become important when distributing the processes across remote networks, because we may not know the type of platform on which our process is running—e.g., UNIX binary does not run on a Linux platform. These materials should be placed into a uniquely named Subversion repository. This is a reasonable request, since good software engineering practices mandate that all project materials should be placed into a versioning control system. This collection of project-specific materials is called the *project repository*.

The project repository contains four types of items:

- *Project Source Code (Required)* – All of the files needed to interpret, compile, link, and/or execute an experimental trial. This can include source code, java class files, or any required libraries to name a few. After the project is successfully built, there should be a program capable of accepting either the name of a data file (containing runtime parameters) or a list of command-line runtime parameters.
- *Configuration Scripts (Optional)* – Sometimes when compiling a project from source it is important to set platform-specific configuration options. Commander client asserts that after the successful conclusion of any configuration scripts, the project should be ready to execute. This can be accomplished through the usage of established tools like autoconf, GNU Makefile, or Apache ANT. One requirement is placed on the configuration scripts – they are not allowed to alter the client computer's file system outside of the directory in which the configuration script is located. This is important because system administrators of various remote nodes might give Commander clients different access permissions.

- *Validation Script (Optional*) – After a trial has finished execution on a remote client, it is important to make sure that the data it generated was correct. When operating on a heterogeneous (and sometimes hostile) network, trials may never finish as local users can kill the processes or nodes may halt or run out of memory. It is possible to have the Commander host check for data errors, but it is better to have the numerous clients each perform their own validation procedure on any locally generated data. These scripts are discussed at length in Section 3.2.
- *Analysis Measures (Optional)* – Commander can automatically perform basic types of analysis operations. Some information is needed from the user, such as what types of graphs to generate (e.g., 2D lines, 3D surface plot), what parameters to use to partition the data set (to construct multiple graphs), and what measure functions to use. These options are discussed in Section 3.3. On a side note, all experimental data presented in Section 4 was generated by the automatic execution of Commander analysis measures.

For most projects, the project repository would not drastically change between experiments. The key differentiator between experiments lies in values for parameter sweeps. Users define sets of parameters to send to a program while creating an *Experiment Builder* file. Experimental parameters can be of three types:

- *Option* – This type of parameter has no associated value. Every datapoint created has this option and thus every trial that is executed runs with this parameter. An example of a command line option from MGP (a genetic programming engine) could be "mgp -useLowMem", which tells the engine to make tradeoffs favoring a small memory footprint over execution speed. Regardless of any other parameters subsequently issued, each trial that is conducted by Commander would subsequently be run with the useLowMem option.
- *Constant* – This type of parameter has a value that is constant for every datapoint in the experiment. An example from MGP could be "mgp –maxGenerations 200".
- *List* – These are the types of parameters over which we want to sweep. These are like constants in that the parameterized type has a value, but unlike the constant parameters, different datapoints can have different parameter values. For example, sweeping over population size {1000,5000} in MGP would generate two unique datapoints with the command lines "mgp –popSize 1000" and "mgp –popSize 5000".

When creating experiments, Commander creates one datapoint for each element in the Cartesian product of the List-type values.

Table 1. Sample parameter types in an Experiment Builder file. The experiment contains 4 datapoints, representing the Cartesian product of "popSize" and "selectionMethod".

Type	Name	Value
Option	"-useLowMem"	
Constant	"-maxGenerations"	200
List	"-popSize"	[1000, 5000]
List	"-selectionMethod"	["Tourn", "Roulette"]

Next, a set of trial packages is created for each datapoint. In fully deterministic experiments, only one trial is needed per datapoint, thus only one trial package is created for each datapoint. However, in most experiments it is necessary to create a number of trial packages for each datapoint due to the nondeterministic behavior of the simulation being studied. After the user specifies the number of *trials per datapoint*, Commander adds trial packages to all datapoints, each with a unique RNG seed. Referring to Table 1, there is a parameter sweep encoding for 4 datapoints. If the number of trials per datapoint is 10, then Commander would create 40 trial packages, each with unique RNG seeds. Commander manages RNG seeds automatically, so if at a later date this experiment is rerun, there is no danger that Commander would accidentally reassign the same RNG seeds. However, experiment files are saved for reuse and Commander does allow the reuse of old RNG seeds at the user's option.

Finally, the user specifies values for Subversion repository name (representing the address of the previously created project repository accessible through a Subversion server), validation script name (representing the name of the validation script in the repository), configure script command (representing the optional configure script in the repository), and the executable name (the resultant binary's name after the configuration script is executed). At this point, the experiment is fully specified through sets of datapoints and trial packages. The experiment is written out to disk for later recovery, and the experiment is transferred to the Commander host to begin task distribution.

3.2 Distributed Architecture

As previously stated, Commander uses a *master-worker* architecture. Incomplete tasks reside in a list on the master and are checked out in arbitrary order by active clients. The clients complete the tasks and transfer results back to the master, which removes completed tasks from its list. The clients continue by checking out new tasks until no tasks remain. Some types of parallel computation feature coherent tasks—types that require partial orderings on task completion. This is generally not true for parameter sweeps, which are usually decoherent in nature. For decoherent tasks, the master-worker architecture is quite well suited, because it maximizes the efficiency of the workers—as soon as they complete one task, they start a new one. The master does not need a complex scheduling algorithm, which allows it to be more efficient at important client requests.

The distributed computation begins with the launch of Commander clients. These Python programs can be started either manually, through scripts on cluster machines, or through Grid interfaces. Clients first conduct an initialization procedure, which includes creating a temporary directory somewhere on the node's file system where project repositories and trial data are stored. The location of this directory is machine specific, and can depend on how the client was installed on the machine. For example, the default on UNIX machines is to use something like *"/tmp/.__cmdr/"* as the working directory.

When ready to begin executing trials, clients connect to the host using an XML-RPC protocol and request a trial package. The package that the host returns is capable of fully specifying everything the client needs to do to successfully run the trial. The client extracts the location of the Subversion repository from the trial package, and uses its local Subversion client to download the project repository to a subdirectory of its working directory.

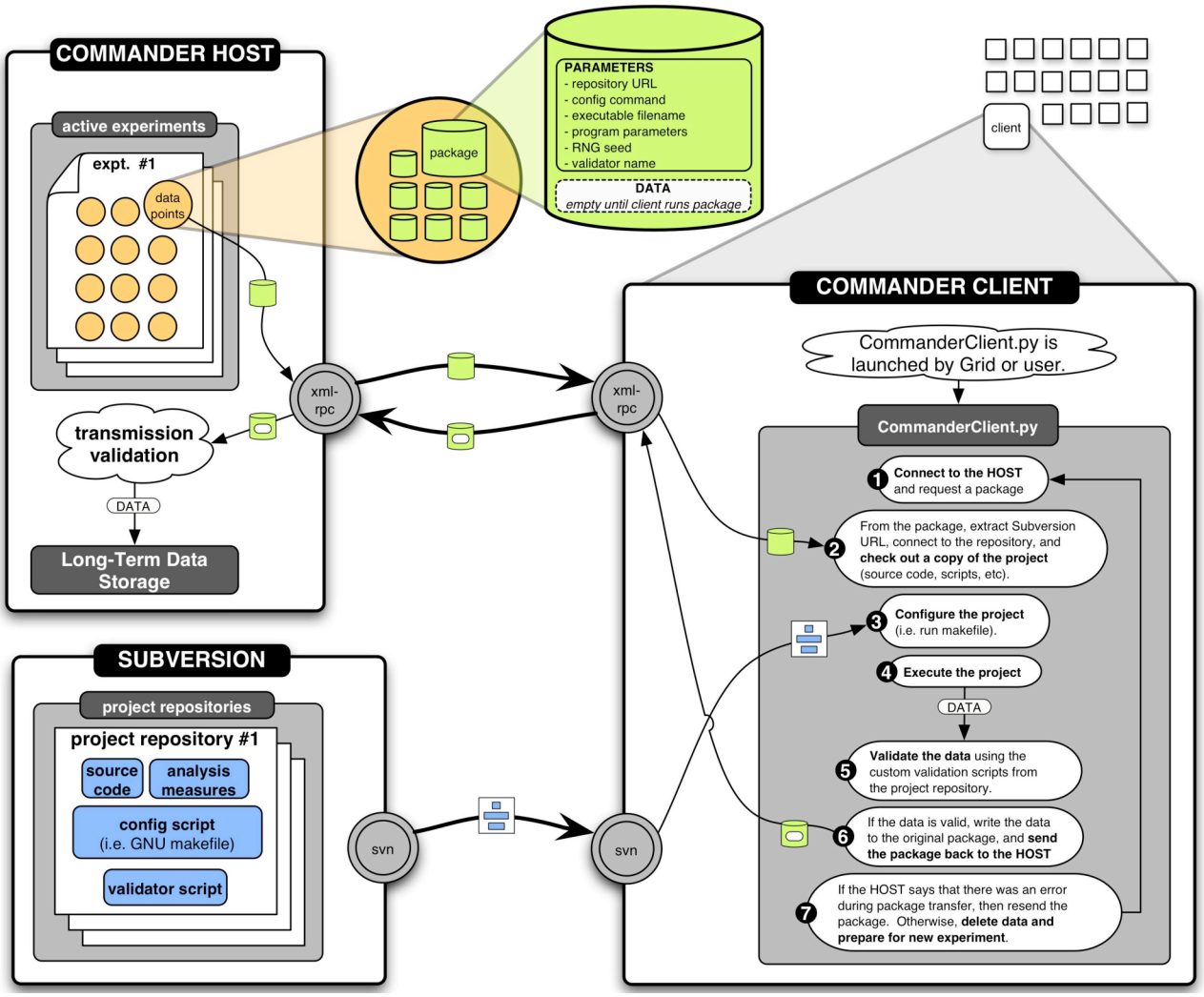

Figure 1. Commander architectural diagram. Commander Clients connect to Commander Hosts and Subversion servers to install, configure, and run experiments. Data is locally validated by the Client and returned to the Host for data reduction and analysis.

Descending into the project directory, the Commander client executes configuration scripts as determined by the trial package in order to prepare the project for subsequent execution. This can include, for example, the execution of *autoconf* scripts to correctly set up platform specific settings and the use of *GNU Makefile* scripts to compile source objects into binary objects. If the binary object needs to accept a file with parameters (as opposed to accepting them listed on the command line), the configuration script should also create that data file from the parameters at this stage. The client then executes the program in a new process, instructing the program to write all data output to a specific data directory. The Commander client then suspends itself until the new process completes.

After the user's program completes, the Commander Client moves to the data directory and uses the optional validation scripts included in the project repository to determine if the data generated is acceptable to return to the host. This is important because there are a number of ways that the new process could fail—most likely through running out of memory or through local user kill signals, and the failed status of a process may not always be made known. If the data passes validation, any files present are recursively compressed and placed into a tarball. The client calculates an MD5 checksum based on this conglomerate file, and adds the tarball along with the checksum information to the trial package. The trial package is transferred back to the host, which inspects both the tarball and the checksum to ensure that no transmission errors occurred.

If the transfer was successful, the client deletes all local copies of the data, but retains the project repository since it is highly likely that it would need it again for another trial. The client continues the process of *check out, configure, execute, test, check in* until no more experiments remain. The client then shuts down, removing all working directories to leave the file system in its original state. Meanwhile, the host, having received the trial package and verified the MD5 sum for correctness against the data, writes the data to a long-term storage directory.

3.3 Data Analysis

When no more incomplete packages for a given experiment remain, Commander accesses the Experiment Builder to determine what types of analysis are desired. It partitions the datapoints through a user-defined equivalence relation and places

each equivalence class in its own graph. Commander then obtains the analysis functions from the repository and applies them to each class of datapoints to generate sets of tuples for each graph.

To rely on an earlier example from Table 1, we could choose "Selection Method" as a partition parameter, "Population Size" as an independent parameter, and "average best of trial fitness" as our measure function. Commander would then use the partition parameter to create two graphs: one for trials using tournament selection and one for trials using fitness proportionate selection. Each graph would be a 2D plot with population size (the independent parameter) on the horizontal axis and the results from "average best of trial fitness" on the vertical axis. The combinations of graph points are represented as tuples and saved in standard formats. Standard viewers for high-quality printing can then use these files.

In this manner, when a project has its own measure functions, there is no longer a need for the user to invest time in writing new scripts to perform data analysis. The researcher merely chooses the parameters with which to partition the graphs, the variables types that serve as inputs to a measure function, and the identity of the measure function. MGP, a GP engine developed at the University of Michigan, comes with a standard suite of measure functions, thus reducing the amount of time required to spend on data analysis. This also gives researchers a much faster turn-around time between experimentation and analysis, allowing quicker results response times. Novel analysis methods must naturally be implemented first by hand and added to the repository, but the process of performing analysis is no longer requisitely tedious.

4. PARAMETER SWEEP EXAMPLES

In this section, we provide useful demonstrations of Experiment Construction, Builder Files, and Data Analysis through common GP experiments involving selection and replacement methods. A number of authors [13,18,21] have studied selection and replacement methods from a mathematical perspective. Discussions of fitness distributions, loss of diversity, and ordinary differential approximations influence our understanding of these dynamics, but very few empirical studies have compared different strategies [11,12]. A number of authors have argued that maintaining genetic diversity is important to EC populations [e.g., 23], but the effects of correlating selection and replacement methods is not known. For example, if both tournament selection and steady-state replacement individually lead to genetic drift, then what are the effects of their combination? What about a steady-state algorithm using fitness-proportionate selection?

To demonstrate that parameter sweeps can augment a researcher's big-picture view, we studied two well-known GP problems while sweeping across various selection and replacement methods. We chose to study 6-input multiplexer and 4-bit parity because these problems have been shown to be tunably-difficult under varying population size and number of generations [e.g., 16]. We configured 6-input multiplexer to use logical NAND and NOR as functions, while 4-bit parity used AND, OR, NAND, and NOR.

Given the relatively few empirical studies comparing selection and replacement strategies, we decided to analyze both parity and multiplexer using different combinations of tournament selection, fitness proportionate selection, generational replacement, and steady-state replacement. We used the MGP genetic programming engine because it was designed to support these command-line parameter configurations. Note that since MGP receives all trial parameters from Commander, we are able to use Commander's Experiment Builder files to describe the experiment here since they fully specify the experiment's parameters to MGP as described in Section 3.1.

Table 2. The parameters in an Experiment Builder file used for sweeping 6-input multiplexer and 6-bit parity.

Type	Field Name	Value
Constant	Max tree depth	512
List	Population Size	$\{2^x : 2 \le x \le 11, x \in Z\}$
List	Max Generations	$\{2^x : 0 \le x \le 9, x \in Z\}$
List	Selection Method	{"Tourn", "Fit Prop"}
List	Replacement Method	{"Generational", "steady-state"}
Cmdr. Setting	Trials per datapoint	100

Commander swept over 10 different population sizes, 10 different maximum generation counts, 2 selection methods, 2 replacement methods, and 2 problems, constructing 800 datapoints representing the Cartesian product of the List parameters. Since there were 100 trials per datapoint, it constructed 80,000 total trial packages. These trials used ~2 CPU-months and finished, with complete analysis, in under 3 days using approximately 50 machines.

The results presented in Figure 2 were generated by Commander analysis scripts and then imported into Igor. Inside our MGP analysis script file was a function, $\text{SuccessCount} : \text{DataPoint} \rightarrow Z$, which examines a datapoint's trial packages and reports the number of successes found with the datapoint's configuration. To generate four graphs, Commander partitioned the set of all datapoints using selection and replacement methods. Each graph subsequently had a different combination of selection and replacement. The graphs were written to files as lists of tuples with descriptive axis titles. Because surface plots can occasionally make comparisons difficult, Table 3 contains the total number of successful trials for each configuration class of selection and replacement. Note that these configurations can be properly ordered by their solution counts.

Figure 2 represents the total number of successes with a more fine-grained analysis, in which number of successes is measured as a function of population size, max generations, and selection and replacement method.

Table 3. Total number of successes out of 10,000 possible trial successes is summed for each configuration class and presented for both 4-bit parity and 6-input multiplexer. The configurations exhibit the same ordering using both problems.

Configuration	Parity	Multiplexer
Tournament Generational	1066	1739
Tournament Steady State	780	1609
Proportionate Steady State	43	76
Proportionate Generational	4	21

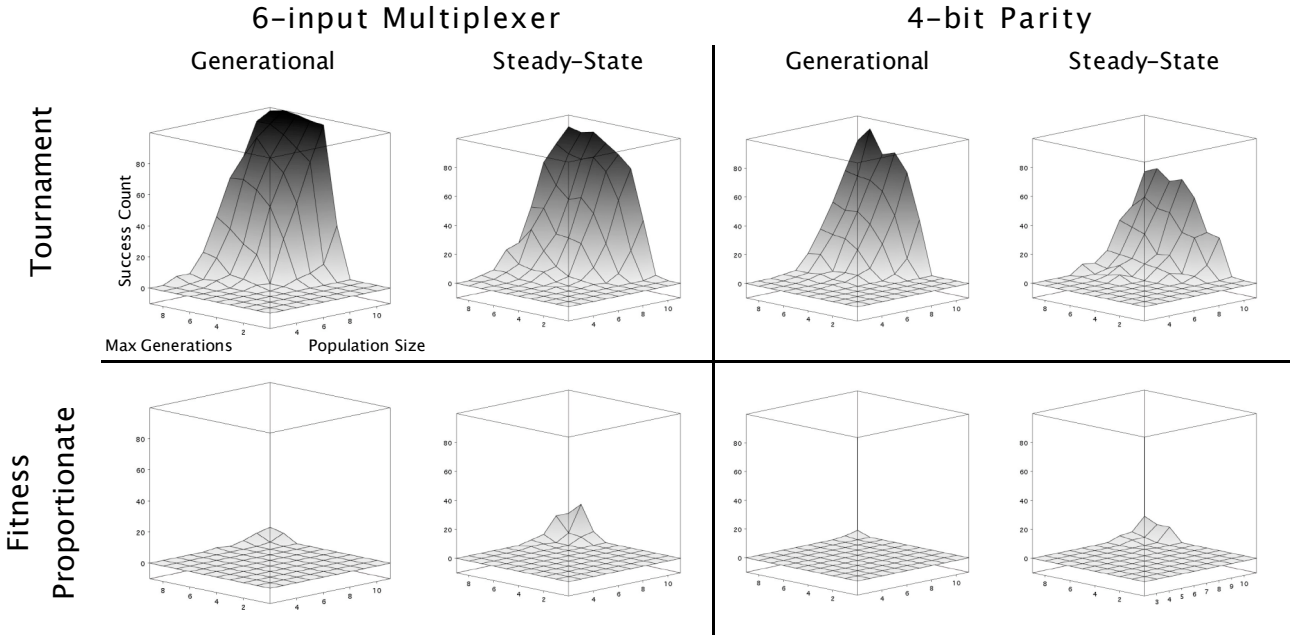

Figure 2. Number of successes as a function of Max Generations, Population Size, and Selection and Replacement method. Max Generations and Population size are represented logarithmically with base 2. All axes are similarly oriented.

5. DISCUSSION

Observation: *There is an ordering on configurations involving selection and replacement methods.*

It is beyond the scope of this paper to analyze this ordered effect, but it serves as an example of the necessity in conducting parameter sweeps. These results have bearing on experimentalists for two reasons.

First, *the nonlinearity in parameter-orderings implies that experimentalists should test many configurations to see a relevant big picture*. For both problems we tested, tournament selection outperformed proportionate selection, but a similar statement cannot be made for replacement methods. When paired with tournament selection, a generational algorithm performed better. However, when a proportionate selection was used, steady state performed better than generational. Experimentalists who assume that parameters exert linear influences on the results would be surprised to learn that a generational replacement has a beneficial effect on success when tournament selection is used, but a harmful effect when proportionate selection is used. These nonlinearities imply that experimentalists should test multiple configurations when constructing a big-picture view. Parameter sweeps can find and classify these types of nonlinearities.

Secondly, *experimentalists should reconsider their notions of significant improvements in GP*. The differences in success between the configurations tested here were several orders of magnitude. Small improvements might not be statistically significant when compared to orders of magnitude change that could be uncovered using a parameter sweep. Further, any benefits gained might be dependent on a narrow configuration setting and thus could be negated by a different configuration.

We are aware that these suggestions for improved GP methodology involve significant work with standard tools. Lengthy scripts are subsequently written to govern and analyze experiments, and experiments often take many more man-hours than before. In an effort to reduce these prohibitive effects, we have presented a program, Commander, to automatically perform many of the tedious tasks involved with building, running, and analyzing distributed parameter sweep experiments. The process of constructing and analyzing experiments should not be tedious. Likewise, there is no reason that distributing decoherent tasks across a network should be difficult. We hope that tools like Commander would enable and encourage the community to perform large parameter sweep. Commander was intentionally designed to easily work with numerous other engines (e.g., ECJ [14], lilgp [24]). Although we frequently use GP engines as an example, we suggest that other researchers in the genetic and evolutionary computation community should use a parameter sweep methodology like the one we have offered in this paper.

6. CONCLUSIONS

Parameter sweep experiments are useful. The presence of nonlinearities inherent to the lower-level interactions of GP means that small experiments can yield an incomplete view of GP dynamics. With large parameter sweeps, experimentalists can examine multiple configurations of influential settings. This process can lead to a more complete mapping of configurations to results and a deeper understanding of GP.

Parameter sweeps are difficult. This methodology requires an engine that can accept parameters and a script that can iterate over collections of parameters. Actually conducting these experiments is expensive in terms of both human labor —writing scripts— and computation —many CPU-hours. The increased number of trials allows for a greater chance that errors can occur which could invalidate results. Further detection and resolution of these errors requires more scripts. Distributed computation can alleviate the pains of lengthy trials, but introduces software engineering

concerns —distribution algorithms, robust data collection— upon which experimentalists should not focus.

Commander is our solution to a generic parameter sweep engine for experimentalists. Commander is designed to automate as much of the experimentation process as possible by alleviating tedious human tasks, providing robust remote trial distribution, and ensuring validity in data collection. Through simple interfaces, Commander allows researchers to quickly define new experiments, run them, and use included analysis measures to see the results. This low turnaround time means that experimentalists can be more productive with their experiments. Data can be interpreted, questions can be asked, and science can be achieved in a productive and responsive fashion.

7. FUTURE WORK

There are a number of ways in which automated experimentation can progress. One method is an automatic-exploration of program parameters. An experimentalist could define the types of parameters over which Commander would sweep. Commander would then attempt to generate a mapping of the configurations to results in meaningful ways. For example, it might automatically explore the interaction of certain list parameters, such as population size in genetic programming.

On a similar thought, GP experimentalists have recently been discussing the number of trials that is necessary to obtain a degree of confidence in results. From mathematics we know that the required number of samples depends on the variance of the random variable being sampled. A future version of Commander could take variance information into account and sample different datapoints with different numbers of trials, achieving a constant degree of results confidence over the entire dataset.

Finally, we're very interested in developing decentralized computing through shared resources. Grid computing can be very expensive, and the development of a local cluster might not be logical because experimentalists are not constantly running experiments. With distribution systems like Commander it is possible for experimentalists to run a Commander client during their CPU's idle time and share their CPU with the larger community. They could accumulate points for participation that could then be cashed in and used on Commander's network. In this fashion, a researcher with low CPU resources could accumulate the ability to run time-intensive trials in small amounts of wall time.

Commander is available online with documentation at http://lattice.engin.umich.edu/Commander

8. ACKNOWLEDGMENTS

Many thanks to Paul Chiusano for helpful comments and for reviewing an earlier draft of this paper. The first and second authors gratefully acknowledge I. Kristo.

9. REFERENCES

[1] Abramson, D., R. Sosic, J. Giddy, and M. Cope. *The laboratory bench: Distributed computing for parametised simulations.* In 1994 Parallel Computing and Transputers Conference, Wollongong, Australia, 1994.

[2] Axelrod, R., *The Dissemination of Culture: A Model with Local Convergence and Global Polarization*, 1997.

[3] Basney, J. et al. *Harnessing the Capacity of Computational Grids for High Energy Physics*. In CCHENP, 2000. Padova, Italy, 2000.

[4] Belding, T. http://drone.sourceforge.net

[5] BOINC. http://boinc.berkeley.edu

[6] Casanova, H., T. Bartol, J. Stiles, and F. Berman, "*Distributing MCell Simulations on the Grid*," Int'l J. High Performance Computing Applications, vol. 14, no. 3, 2001.

[7] Casanova, H., G. Obertelli, F. Bermand, and R. Wolski. *The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid*. In Proceedings of SC00, November 2000.

[8] Collins-Sussman, B., B. Fitzpatrick & C. Pilato. *Version Control With Subversion*. svnbook.red-bean.com 2004.

[9] Daida et *al*. *Challenges with Verification, Repeatability, and Meaningful Comparisons in GP*. In GP97, Koza et al. (Eds). Morgan Kaufmann. 1997.

[10] Daida et al. *Challenges with Verification, Repeatability, and Meaningful Comparison in GP: Gibson's Magic*. In GECCO 1999. San Francisco, CA: Morgan Kaufmann. 1999.

[11] Daida, J. et *al*., *Visualizing the Loss of Diversity in GP* in CEC 2004. Piscataway: IEEE Press, 2004.

[12] Daida, J., Samples, M., et al. *Demonstrating Constraints to Diversity with a Tunably Difficult Problem for GP*. In CEC 2004. Piscataway: IEEE Press, 2004.

[13] Goldberg, D. & K. Deb, "*A comparative analysis of selection scheme used in genetic algorithms*," in FOGA'91, Rawlins Ed. San Mateo, CA: Morgan Kaufman, 1991.

[14] Luke, S. ECJ, George Mason University, ECLab, Fairfax 2004.

[15] Luke, S. and Panait, L. *Is the Perfect the Enemy of the Good*? In GECCO 2003. Spring-Verlag, Berlin, 2003.

[16] Luke, S. and L. Spector. 1998. *A revised comparison of crossover and mutation in GP*. In GP98: J. Koza et al, eds. 208--213. San Francisco: Morgan Kaufmann.

[17] Meysenburg, M. M., & Foster, J. A. *Random generator quality and GP performance*. In Banzhaf, W. et al. (Eds). In GECCO 1999. San Francisco, CA: Morgan Kaufmann Publishers.

[18] Motoki, T. *Calculating the Expected Loss of Diversity of Selection Schemes*. Evolutionary Computation 10(4): 397-422 (2002).

[19] Paterson, N. and Livesey, M. *Performance Comparison in GP*. In *LPB GECCO 2002*. San Francisco, CA: Morgan Kaufmann 2002.

[20] Spice. bwrc.eecs.berkeley.edu/Classes/IcBook/SPICE/

[21] Syswerda, G. *A Study of Reproduction in Generational and Steady State Genetic Algorithms,* in FOGA'91, Rawlings ed., pp. 94-- 101, San Mateo: Morgan Kaufmann, 1991.

[22] Thain D., and M. Livny, *Building Reliable Clients and Servers*, in Foster et al. (EDS), *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, 2003.

[23] Ursem, R.K. *Diversity-Guided Evolutionary Algorithms*. In PPSN-2002, pages 462-71, Springer-Verlag. 2002.

[24] Zongker, D. and Punch, W. lilgp, Michigan State University, GA Research and Applications Group, Lansing 1995.