The SEEDS Platform for Evolutionary and Ecological Simulations

Brian D. Connelly, Luis Zaman, and Philip K. McKinley
BEACON Center for the Study of Evolution in Action
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI USA
bdc@msu.edu

ABSTRACT

Over the past few decades, evolutionary computation (EC) has grown substantially in use for biologists and engineers alike. Its transparency makes it an indispensable tool for studying evolutionary and ecological dynamics, and it has provided researchers with new insights that would be tremendously difficult, if not impossible, to gain using natural systems. In addition, EC has proven to be a powerful search algorithm for engineering applications, and has produced numerous novel and human-competitive solutions to complex problems.

Although several well-established packages are readily available, it seems that when most users harness the power of evolutionary computation, they do so using "home-grown" solutions. This can likely be attributed to the ease with which simple models are created, the user's need for customization, and the sizeable learning barrier imposed by available solutions, as well as difficulties in extending them.

We present SEEDS, a modular, open-source platform for conducting evolutionary computation experiments. SEEDS provides a simple, flexible, and extensible foundation that enables users with minimal programming experience to perform complex evolutionary and ecological simulations without having to first implement core functionality. In addition, SEEDS provides the tools necessary to make sharing data and reproducing experiments both easy and convenient.

Categories and Subject Descriptors

J.3 [**Computer Applications**]: Life and Medical Sciences—*Biology and genetics*; D.m [**Software**]: Miscellaneous; I.6.3 [**Computing Methodologies**]: Simulation and Modeling—*Applications*

General Terms

Design,Theory

GECCO'12 Companion, July 7–11, 2012, Philadelphia, PA, USA.

Keywords

evolution,ecology,simulation,platform

1. INTRODUCTION

Evolutionary computation (EC) focuses on populations of individuals subjected to processes that are the necessary and sufficient conditions for evolution, such as mutation, selection, and inheritance [3]. Through the evolutionary process, individuals become better suited to their environments, which are defined by the user. Individuals can represent anything from candidate solutions to an optimization problem, such as equations fitting a dataset, to natural organisms that interact with each other and their environment. In any instance, evolution can be seen as a guided search process.

Digital evolution and other forms of evolutionary computation have been successfully used to address a number of fundamental questions in biology. (e.g., [12, 10]). In these areas, EC provides benefits that are not easily obtained, or simply impossible, when studying natural systems. Among these is the ability to study populations over thousands or even millions of generations, which is not feasible in most natural species. A second major benefit offered by EC is the complete transparency it provides to researchers, allowing them to observe each individual in the population, as well as its behaviors, interactions, and even its complete genetic encoding. These features offer researchers unique insights into the evolutionary trajectories of behaviors over evolutionary timescales, including how those behaviors evolve, whether they can be maintained, and the abundances at which they exist in populations. Because of these benefits, there is great interest within the biological sciences to use EC techniques to generate and test hypotheses, supplement research using natural systems, and study phenomena that simply cannot be addressed otherwise.

Evolutionary computation has also been successfully used in engineering during the design and optimization processes (e.g, [6, 7]). In many instances, EC has produced novel and human-competitive results [11], cementing its status as a valuable tool throughout engineering.

Although several EC platforms are available, their widespread use is often limited by a number of factors. One of these is a sharp learning curve, which often presents a formidable barrier to those would-be users from biological fields who may lack significant computer programming experience. A second factor is the inflexibility of many platforms, which makes alterations or additions in functionality cumbersome. Because of these and other shortcomings, users

often either create their own EC system, re-implementing standard functionality such as populations and selection, or avoid using EC altogether.

The SEEDS platform is intended to address these problems by providing a foundation upon which researchers can conduct experiments using a wide variety of techniques and models. Because meeting the needs of all researchers would be an impossible task, SEEDS is designed as a modular system, where users can easily modify or extend its capabilities without also having to implement fundamentals. Through its well-defined interfaces and implementation in the Python programming language, researchers of all levels of programming experience are able to both use and extend SEEDS. Additionally, SEEDS places an emphasis on repeatability and data sharing, allowing users to easily recreate experiments and share their models.

To date, SEEDS has been used to explore a wide variety of topics related to evolution and ecology. SEEDS was initially developed in order to study the role spatial structure plays in maintaining diversity [2]. Additional studies have used SEEDS to explore the use and maintenance of horizontal gene transfer [1], the evolution of public goods cooperation, the evolution of cooperation in probabilistic game theoretic models, and speciation and sexual dimorphism resulting from divergent selection in limited resource environments.

The SEEDS distribution includes several prebuilt components that enable experiments to be run immediately after installation. Among these is a model of the Rock-Paper-Scissors (RPS) game—**RPSCell**. Although this common game may seem trivial, it allows users of all ages and backgrounds to both experience and interact with many aspects that affect population dynamics. Such hands-on experimentation can be a powerful tool for learning fundamental processes in evolution and ecology. We will refer to the RPS example throughout the remainder this paper.

We introduce SEEDS as follows. Section 2 presents the design of SEEDS and its organization. Section 3 discusses how experiments are configured and performed. The plugin system and how it can be used to extend the functionality of SEEDS to match the research questions at hand is described in Section 4. Finally, the significant additions planned for near-term inclusion are outlined in Section 5.

2. THE SEEDS PLATFORM

In recent years, the Python programming language has experienced an enormous growth in its use in scientific computing. Part of this growth can be attributed to the fact that, in contrast to many scientific computing environments, Python is a general purpose programming language with a great deal of flexibility [15]. A second reason for Python's success is its broad user community, which has produced several powerful and easily-approachable packages for manipulating, analyzing, and displaying scientific datasets; examples include NumPy [14], SciPy [9], and Matplotlib [8]. These and many other packages are easily obtained through Python's various package management utilities or Python distributions targeted specifically for use in science. Python's ubiquity is another major benefit, with versions available for most operating systems. Finally, Python is open source software, which means it does not impose a financial barrier to adoption. Although Python often does not always match lower-level languages such as C, C++, and Java in performance, especially in numerically-intense tasks, its ease of development and maintenance often more than make up for differences in performance on modern computing resources.

Figure 1 shows the class diagram for SEEDS. A SEEDS Experiment consists of a Population of Cells. The possible phenotypes of a Cell are limited only by its implementation, which allows Cells to capture a wide variety of behaviors and genetic representations. The interactions among Cells are defined by the underlying Topology, an arbitrary graph structure which orients Cells in space. Cells also have access to Resources, which are distributed throughout space using a separate Topology. Finally, Actions allow the user to modify the environment, interact with cells, and produce output data during the Experiment. Each of these classes is a Plugin, and users can easily modify and extend SEEDS through the development of custom Plugins. In the following sections, we describe each of these classes in more detail.

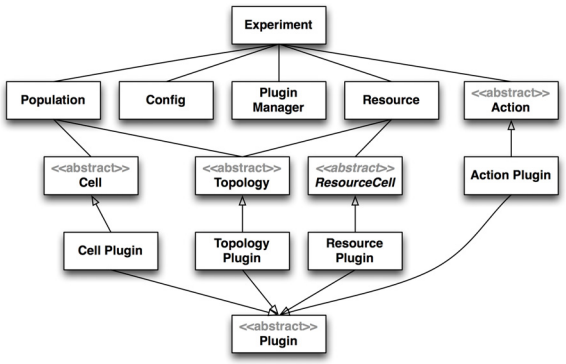

Figure 1: SEEDS Class Diagram

2.1 Experiment

The **Experiment** object controls all aspects of an experiment. This includes managing the configuration, the population, and any available resources. During initialization, the Experiment object loads the specified configuration file and uses this information to create a population of individuals, environmental resources, and any actions scheduled to be performed, such as writing data or interacting with the population or resources, each of which is described below. During each unit of time, or *epoch*, the Experiment object updates the state of the population and resources, and executes any scheduled actions. Upon initialization, each Experiment is given a globally-unique identifier (GUID), allowing experiment data to be easily catalogued.

2.2 Population

A population consists of a collection of individuals that reside on nodes in a graph structure. The graph defines the potential interactions of individuals. If the nodes in which two individuals reside are connected by an edge, those individuals have the potential to interact with each other, depending on the implementation of their Cell type. The use of arbitrary graphs to define the interactions within a population is one of the most powerful features of SEEDS. Experiments focus on one Population object; however, the interactions within that Population can be defined such that multiple independent subpopulations exist.

When a population is updated, a preconfigured number

of individuals are chosen randomly with replacement, and the state of these individuals is updated based on the rules defined by the corresponding Cell instance. By default, the number of individuals updated is equal to the size of the population, so each individual is expected to be updated during each epoch, on average.

2.3 Cell

In SEEDS, the **Cell** object is used to represent each individual, and is therefore often the primary focus of an Experiment. All Cells must define three methods: `__init__` (a constructor), `update`, and `teardown`. The constructor initializes all properties associated with that object, perhaps reading values from the configuration file. The `update` method is intended to update the state of that organism, and is executed during each epoch of the experiment, on average. The update method may cause the Cell to reproduce, to consume resource, to produce some resource, to interact with a neighboring Cell, to die, or be subject to any other event pertinent to an individual in the simulation. During each epoch, a subset of Cells from the population is chosen randomly and updated using this method. Finally, the `teardown` method is called whenever a Cell is removed from the population. This method is intended to handle any cleanup tasks necessary, such as the closing of files or the freeing of references. These methods are defined in the Cell base class. Any Cell object must be a subclass of this class, which also provides methods for retrieving and managing the neighbor list of each cell in the population.

A SEEDS cell is agnostic with respect to any particular evolutionary algorithm. What defines an individual is decided by the user: its representation, its behaviors and capabilities, its interactions, and ultimately its fitness. By creating a Cell subclass, users can implement a Cell class to suit their particular needs. For example, each cell could represent a state, such as alive or dead, as in Conway's Game of Life [4]. Alternately, each cell could encapsulate an artificial neural network, such as those evolved using NEAT [16]. A cell could even represent a program and have virtual hardware associated with it, as in the Avida system [13].

Continuing our example, RPSCell is a cellular automaton model. Each Cell in the Population plays either the Rock, Paper, or Scissors strategy. When a Rock cell encounters a Paper cell, it becomes a Paper cell, because Paper covers Rock. However, if that Rock cell encountered a Scissors cell, it would remain a Rock, because Rock is not beaten by Scissors.

2.4 Topology

The interactions of individuals within a population are defined by their **Topology**. In many forms of evolutionary computation, the interactions among individuals are defined by a lattice, where an individual residing in a node can interact with any of its 4 or 8 neighbors. To allow for more flexibility, SEEDS models interactions using arbitrary graphs. As with lattices, each individual occupies a node in the graph, and edges between two nodes represent a potential interaction between those corresponding individuals. The distribution of these edges, however, need not be regular as in a lattice, but can instead define more complex interaction networks. This includes disconnected graphs, where certain subsets of the population do not interact with others.

Graphs are created and maintained using NetworkX [5], which provides a vast library of graph generators and allows for the creation of custom graph structures. Additionally, NetworkX offers functions that calculate many common graph metrics, which aid in understanding and comparing graphs.

Several topologies are included with the SEEDS distribution. As one example, the CartesianTopology topology is created by randomly placing nodes in a unit Cartesian plane and adding edges between nodes that are within a distance yielding an expected number of neighbors per node [2]. The CartesianTopology allows neighborhood sizes to grow linearly, instead of the geometric increase seen in lattice models as radius is increased.

Topologies are created by subclassing the *Topology* class, which provides a number of methods for maintaining the structure of the graph. Each topology must implement the `__init__` constructor method, which creates the appropriate graph, and the `teardown` method, which performs any necessary tasks before a topology is deleted. Topologies that support changes in structure during an experiment can additionally implement the `add_node`, `remove_node`, `add_edge`, and `remove_edge` methods, which handle the necessary changes to the graph.

2.5 Config

The **Config** object manages the configuration for an Experiment. By wrapping Python's ConfigParser module, the Config object organizes the configuration into sections, one for each Experiment, Population, Cell, Resource, Topology, and Action. Each configuration section contains a set of parameter-value pairs, which define the value for a property in the respective object. These values can represent parameters for the object or define how that object behaves.

Typically, an Experiment will create a configuration from an input file. Cells such as RPSCell, for example, can query the Config object to obtain values to be used from the **RPSCell** section. RPSCell includes the *distance_dependent* parameter, which takes a boolean value specifying whether or not cell interaction is directly proportional to their proximity. When querying the Config object, default values can also be specified for use when that parameter is unspecified in the configuration file.

The portion of an example configuration file that deals with RPSCell is shown in Listing 1. This configuration will be expanded upon in Section 3.3.

```
[RPSCell]
distance_dependent = True
```

Listing 1: Segment of a configuration file that defines the behavior of RPSCells in a Population. Here, cells will be more likely to interact with nearby cells.

The Config object can also export an Experiment's current configuration. The resulting configuration file includes all parameters defined in the original configuration file, as well as all default values used and the seed for the random number generator. This feature greatly aids in reproducing experiments.

2.6 Resource

Individuals can interact with each other and their environment through the consumption and production of resources. As shown in Figure 2, each **Resource** specifies

its own topology, which allows the distribution for each resource to be controlled. Each node in a resource contains a *ResourceCell* object, which defines the resource at that point in space. For example, SEEDS's *NormalResource* ResourceCell defines a resource as a level, an inflow, a decay, and a diffusion. The level defines the amount of resource currently present in that cell. Inflow defines the rate at which new resource enters that cell. Decay and diffusion result in resource being lost from the environment at a configured rate and resource flowing into neighboring ResourceCells at a configured rate, respectively.

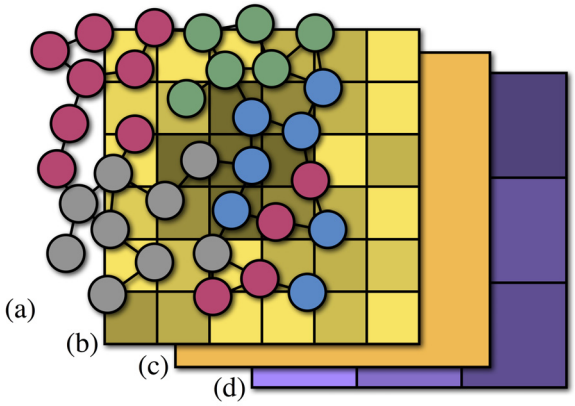

Figure 2: Population and resource topologies. Each exists in the same unit Cartesian space; however, they partition that space differently. (a) The population topology is independent from the three resource topologies (b-d). The resource in (b) partitions the resource as a 6×6 lattice, while the resource in (c) is global, and resource in (d) partitions the resource topology as a 3×3 lattice.

The SEEDS distribution also contains the *SineResource* and *SquareResource* types, which vary the level of resource in a given cell according to a sine and square function, respectively. By subclassing the *ResourceCell* class, new resource types can easily be defined.

New resources must implement three common methods. The `__init__` constructor method, which is executed when a ResourceCell is created, initializes the object and sets the initial level as configured. The `update` method adjusts the level of the resource according to the inflow, decay, and diffusion properties of that ResourceCell. Finally, the `teardown` method allows the ResourceCell to clean up its state before being deleted.

When a cell interacts with a resource, its coordinates in unit Cartesian space are projected onto the resource's topology. This interaction will then affect the level of the ResourceCell nearest to that coordinate. Therefore, the extent to which individuals share a particular resource pool depends on the partitioning of that resource's ResourceCells. A topology with fewer nodes will likely experience more overall competition for resource than a topology with more nodes. Although Resources use the same topology classes as do populations, they are currently limited to lattices. This restriction will be lifted in a future release.

2.7 Action

*Action*s define events that occur at specified times during an experiment. An action can affect any apsect of the experiment, from the population and its individuals to resources. Actions are also the primary way in which output data are produced. Some examples of actions included with SEEDS are listed in Table 1.

Table 1: Sample Actions Included with SEEDS

Action	Description
PrintExperimentInformation	Print detailed information about the experiment and the software environment
PrintCellTypeCount	Print the abundances of each cell type in the population
SetResourceAvailability	Toggle the availability of a resource
StopOnConvergence	Stop the experiment when the number of different types of cells in the population falls below a threshold

As an example, the *PrintCellTypeCount* action creates a comma-separated values (csv) file containing the number of each cell type present in the population at that time. These data can be used for further analysis following the experiment, or plotted, as demonstrated in Figure 3. The *PrintCellLocations* action writes a csv file containing the location of each cell in the population, which can be used to visualize the distribution of strategies in space, as shown in Figure 4.

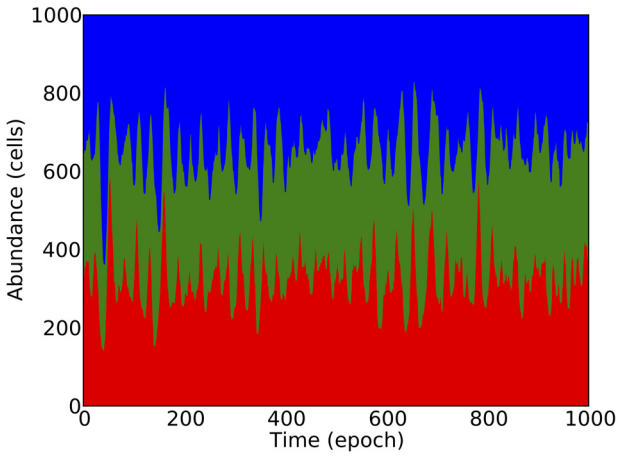

Figure 3: Abundances of Rock (red), Paper (green), and Scissors (blue) cells over time in a population containing 1,000 RPSCell cells

All actions inherit from the *Action* class. Like most other SEEDS classes, actions must implement `__init__`, `update`, and `teardown` methods, which are executed when instances of that action are created, updated, and deleted, respectively. Each action also specifies an *epoch_start*, *epoch_end*, and *frequency*, which define when the action begins, when it stops, and how frequently it occurs within that window of time, respectively. Plots similar to those shown in Figures 3 and 4 can be produced through the creation of Actions that plot population data during or after an Experiment. Several Actions that create plots are contained in the `contrib` directory in the SEEDS distribution, including the *PlotCellTypeStack* and *DrawPopulation* actions used to create Figures 3 and 4, respectively.

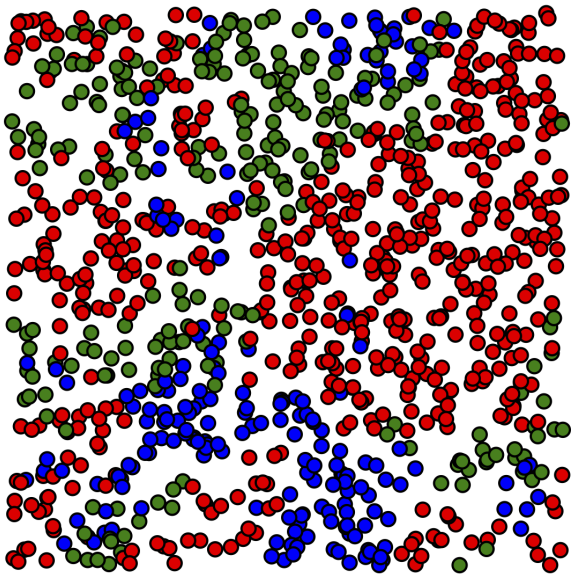

Figure 4: Snapshot of a population of 1,000 Rock (red), Paper (green), and Scissors (blue) cells during an experiment using the CartesianTopology. In this example, each cell interacts with its 10 nearest neighbors, on average.

2.8 Plugin

In SEEDS, each custom Cell, Topology, ResourceCell, and Action is also an instance of the **Plugin** class. These Plugins, described in detail in Section 4 allow the functionality of SEEDS to be extended without recompilation or modification of the SEEDS installation. Plugins also contain version information and other metadata, which allow for experiments to be replicated exactly.

2.9 Plugin Manager

The **PluginManager** object receives and handles requests for Cell, Topology, Resource, and Action Plugins. The plugin manager scans the Plugins available in the directories specified by the configuration. If the plugin (and version) specified is available, an object of that type will be returned. Otherwise, an exception will be raised.

3. USING SEEDS

This section describes how SEEDS is typically used. First, Section 3.1 discusses how SEEDS can be obtained and installed. Section 3.2 details how experiments can be performed. Finally, Section 3.3 introduces the SEEDS configuration file, which can be used to define a single experiment or family of experiments.

3.1 Obtaining and Installing

SEEDS is open-source software, released under the Apache License 2.0[1], and is publicly available via a number of channels. The most straightforward way to install SEEDS is using the `pip` or `easy_install` tools. Alternately, all versions of SEEDS are available on the SEEDS GitHub page[2], which also contains documentation, issue tracking, and development history. SEEDS can be installed from source using the standard Python Distribution Utilities.

SEEDS is designed to have minimal dependencies. A working installation will require Python 2.6.5 or greater (including 3.2) and NetworkX [5]. Although not required, SciPy [9], NumPy [14], and Matplotlib [8] are also recommended, due to their frequent use in third-party plugins for analysis and plotting.

3.2 Running an Experiment

The most common way to conduct experiments is using the `runseeds.py` script. This script is installed with SEEDS, and includes a number of command-line options for configuring and running experiments.

Typically, a user will create a directory in which the experiment will be managed. That directory contains a configuration file, further described below, and an optional subdirectory containing any plugins to be used.

Experiments can also be run from within a Python interpreter or another script. Doing so simply requires creating an Experiment object and iterating over that object. A simple example of this is shown in Listing 2. This flexibility allows experiments to easily be run in a number of ways, from command line tools and graphical user interfaces to web-based apps.

```
import seeds as S
experiment = S.Experiment('examples/Rock-
    Paper-Scissors/seeds.cfg')

for epoch in experiment:
    print('Epoch {e} done'.format(e=epoch))
```

Listing 2: Python code to create a SEEDS experiment, load the configuration file for the Rock-Paper-Scissors experiment, and perform the experiment.

3.3 Configuration

Configuration files define all aspects of an experiment, from the duration of the experiment, the Cell type and Topology to use, which resources are defined, which actions to run, and where to place resulting data.

As shown in Listing 3, a configuration file places each configurable item in its own section, indicated by brackets. In this example, the experiment, as defined in the **Experiment** section, will run for 1,000 epochs. It will use the *PrintCellTypeCount* and *StopOnConvergence* actions, each of which is configured below. It will find any third-party plugins in the *plugins* and *customcells* directories. Finally, any data written during this experiment will be placed in the *data* directory.

The population, defined in the **Population** section, will use the *RPSCell* cell type connected using the *Cartesian Topology* topology. Here, the **:large** label appended to the topology indicates that the experiment will use the configuration specified in the **CartesianTopology:large** section, as opposed to the **CartesianTopology:small** section, which is also defined. This labeling allows a single configuration file to define multiple settings for each item, which simplifies managing the configuration of ensembles of experiments.

[1] http://www.apache.org/licenses/LICENSE-2.0
[2] https://github.com/briandconnelly/seeds

When this experiment is run, the *data* directory will be created. Any actions that produce data will write to this directory. An example is *PrintCellTypeCount*, which writes a comma-separated-values (csv) file containing the abundance of each type of cell for a cellular automaton model.

```
[Experiment]
epochs = 1000
actions = PrintCellTypeCount,
    PrintCellLocations,StopOnConvergence
plugin_dirs = plugins,customcells
data_dir = data

[Population]
cell = RPSCell
topology = CartesianTopology:large

[RPSCell]
distance_dependent = False

[CartesianTopology:small]
size = 2500
periodic = True
expected_neighbors = 10
remove_disconnected = False

[CartesianTopology:large]
size = 250000
periodic = True
expected_neighbors = 10

[PrintCellTypeCount]
start_epoch = 100
frequency = 1

[StopOnConvergence]
threshold = 3
```

Listing 3: Example configuration for the *Rock-Paper-Scissors* experiment in which each cell plays either the Rock, Paper, or Scissors strategy against a randomly-chosen neighbor. This configuration file is available in the SEEDS distribution under `examples/Rock-Paper-Scissors`.

Configuration files, including all default values and random number generator seeds necessary to replicate an experiment, are automatically created by SEEDS when run using the `runseeds.py` script with the `-genconfig` flag.

4. EXTENDING SEEDS

Although the SEEDS distribution includes a large number of Cells, Topologies, Actions, and Resources, it can be extended to address many other types of questions involving interactions within and among populations. Specifically, SEEDS has been designed to provide maximum customization and extensibility through a plugin system.

Plugins allow users to specify the behaviors, the environments, and the resources that best describe their model's needs through the creation of new Cells, Topologies, Resource Cells and Actions. By using plugins, users do not need to delve into SEEDS internals, modify the SEEDS installation, or wait for future releases in order to gain new capabilities. Instead, plugins are created locally and can be integrated into experiments immediately.

Plugins are created by subclassing the **Plugin** class. All plugin classes must define a number of properties that describe the plugin and allow experiments to specify specific versions to be used. These properties are listed in Table 2. Additionally, plugins must implement the `__init__`, `update`, and `teardown` methods, which initialize the plugin, update its state, and perform any necessary cleanup activities, respectively.

Table 2: Properties Defined by Each Plugin

Property	Description
__name__	The name of the plugin
__description__	A detailed description of the plugin
__version__	A tuple containing the major and minor version of this plugin
__author__	The plugin author
__credits__	Additional credits
__type__	An integer specifying the type of the plugin (e.g., Cell, Topology, etc.)
__requirements__	A list of required plugins and their version numbers

All of SEEDS's built-in Cells, Topologies, Resource Cells, and Actions are implemented as plugins located in a system-wide plugin directory. These plugins can be modified or extended by creating local versions of them, allowing users to alter how they use SEEDS without affecting other users of the same SEEDS installation. The SEEDS distribution contains templates to aid in the development of plugins of all types. A number of user-contributed plugins that are likely to be of use to a wider audience are included in the `contrib` directory of the SEEDS distribution.

Our running Rock-Paper-Scissors example is a cellular automaton model. SEEDS Cells, however, can represent other types of models as well. In this example, we will construct a simple genetic algorithm model in which populations of individuals evolve to produce a target string. Here, the "genome" of each individual is an array of characters. At the beginning of the experiment, these genomes are initialized randomly using the 26-character Latin alphabet and the space character. The implementation of this cell is shown in Listing 4. For brevity, comments and error checking are not included.

When a cell is updated, two random neighbors are chosen as parents with probability proportional to their fitness using roulette-wheel selection. A random, two-point crossover is then performed using their genomes, and mutations are applied site-by-site on the recombined genome. This design approximates a tournament selection with tournament size equal to the number of neighbors that each cell has.

5. FUTURE DIRECTIONS

SEEDS is feature-rich, and has been used in several previous and ongoing research projects (e.g., [2, 1]). Nevertheless, a number of additions are planned that will increase repeatability of experiments, allow for easier distribution of plugins and configurations, offer new avenues for storing and interacting with resulting data sets, and make SEEDS a more approachable platform for use in education. This section details the significant enhancements planned for the near future.

5.1 Unit Testing Framework

One of the primary goals for SEEDS is to maintain a dependable platform that is backed by solid software engineering practices. Among these are unit tests, which are cur-

rently being integrated at all levels, from the core modules to plugins. By enabling the platform to be tested from experiment to experiment and version to version, we hope to maintain high levels of reliability and reproducibility.

5.2 Self-Contained Experiments

SEEDS's flexible plugin system allows users to easily extend its functionality; however, this feature might also make sharing experiments that use many custom plugins and configurations more difficult. In order to facilitate sharing among users, a container format is being developed, which will permit users to include all files associated with a given experiment in a single file. These container files will use a compressed, self-contained archive containing a manifest file describing the contents of the archive, one or more configuration files used to recreate an experiment, and all custom plugins used. The SEEDS distribution will include utilities for creating and editing these archives, as well as extracting individual elements from them.

5.3 Flexible File IO

Although the comma-separated values (csv) files created by actions that write data are fairly standard and readable by a wide variety of software packages, greater flexibility in the output formats produced by SEEDS and its actions can allow users to easily incorporate experiments into their own workflow, such as the use of relational databases to merge, query, and analyze data sets. To support such functionality, SEEDS will integrate a modular layer for reading and writing data that allows plugins to be created for different data formats. Methods provided by this layer to read and write data would allow Actions to write data in the format specified by the user.

5.4 Graphical User Interface

Although SEEDS can be used to conduct research in a number of fields, it is also intended as a hands-on learning tool for use in the classroom. Through educational modules containing well-defined experiments, necessary configurations and plugins, as well as additional background information and prompts for exploration, SEEDS has the capability to be a powerful tool for students of all ages to observe and affect fundamental properties of evolution and ecology.

Currently, experiments can only be performed from the command line. In order to reach a wider audience, an easy-to-use graphical user interface (GUI) will be developed to allow users to interact with experiments in a more intuitive way. Although this extension will be a major undertaking, SEEDS's modularity promises to enable the development of first-class interfaces of all kinds.

Acknowledgments

The authors are grateful to Anuraag Pakanati for his feedback. This material is based in part upon work supported by the National Science Foundation under grants DBI-0939454, CNS-1059373, CNS-0915855, CCF-0820220, and CNS-0751155. Luis Zaman was supported by an AT&T Labs Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

6. REFERENCES

[1] B. D. Connelly, L. Zaman, P. K. McKinley, and C. Ofria. Modeling the evolutionary dynamics of plasmids in spatial populations. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 227–233, 2011.

[2] B. D. Connelly, L. Zaman, C. Ofria, and P. K. McKinley. Social structure and the maintenance of biodiversity. In *Proceedings of the 12th International Conference on the Synthesis and Simulation of Living Systems (ALIFE)*, pages 461–468, 2010.

[3] A. Eiben and J. Smith. *Introduction to evolutionary computing*. Springer Verlag, 2003.

[4] M. Gardner. Mathematical games: The fantastic combinations of John Conway's new solitaire game "Life". *Scientific American*, 223:120–123, 1970.

[5] A. A. Hagberg, D. A. Schult, and P. J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference*, pages 11–15, Aug. 2008.

[6] G. Hornby, J. Lohn, and D. Linden. Computer-automated evolution of an X-band antenna for NASA's space technology 5 mission. *Evolutionary Computation*, 19:1–23, 2011.

[7] J. Hu, E. Goodman, S. Li, and R. Rosenberg. Automated synthesis of mechanical vibration absorbers using genetic programming. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 22:207–217, 2008.

[8] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, May-Jun 2007.

[9] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2001.

[10] B. Kerr, M. Riley, M. Feldman, and B. Bohannan. Local dispersal promotes biodiversity in a real-life game of rock-paper-scissors. *Nature*, 418:171–174, 2002.

[11] J. Koza. Human-competitive results produced by genetic programming. *Genetic Programming and Evolvable Machines*, 11:251–284, 2010.

[12] R. E. Lenski, C. Ofria, R. T. Pennock, and C. Adami. The evolutionary origin of complex features. *Nature*, 423:139–144, 2003.

[13] C. Ofria and C. Wilke. Avida: A software platform for research in computational evolutionary biology. *Artificial Life*, 10:191–229, 2004.

[14] T. E. Oliphant. *A Guide to NumPy*, volume 1. Trelgol Publishing, 2006.

[15] F. Pérez, B. E. Granger, and J. D. Hunter. Python: An ecosystem for scientific computing. *Computing in Science and Engineering*, 13:13–21, 2011.

[16] K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10:99–127, 2002.

```
import random
import string

from seeds.Cell import *
from seeds.Plugin import *
from seeds.SEEDSError import *
from seeds.utils.sampling import roulette_select

__name__ = 'SentenceCell'
__description__ = 'Evolving configured target sentence using GAs'
__version__ = (1,0)
__author__ = 'Brian Connelly <bdc@msu.edu>'
__credits__ = 'Brian Connelly and Luis Zaman'
__type__ = 1

class SentenceCell(Cell, Plugin):
    alphabet = string.ascii_lowercase + string.ascii_uppercase + ' '
    types = ['Sentence']

    def __init__(self, experiment, population, node, type=None,
                 name="SentenceCell", label=None):
        super(SentenceCell, self).__init__(experiment, population,
                                           node=node, type=0, name=name, label=label)

        self.target = self.experiment.config.get(self.config_section, 'target')
        self.genome_length = len(self.target)
        self.mutation = self.experiment.config.getfloat(self.config_section, 'mutation',
                                                        default=0)

        self.genome = random.sample(self.alphabet, self.genome_length)
        self.calculate_fitness()

    def update(self):
        self.neighbors = self.get_neighbors()
        neighbor_fitnesses = [n.fitness for n in self.neighbors]

        # Use roulette wheel to find the most fit parents
        parents = roulette_select(items=self.neighbors, fitnesses=neighbor_fitnesses, k=2)

        # Choose random crossover points
        cp1 = random.randint(0, self.genome_length-1)
        cp2 = random.randint(0, self.genome_length-1)

        self.genome[:cp1] = parents[0].genome[:cp1]
        self.genome[cp1:cp2] = parents[1].genome[cp1:cp2]
        self.genome[cp2:] = parents[0].genome[cp2:]

        # Apply mutations to the offspring
        for i in range(self.genome_length):
            if random.random() < self.mutation:
                self.genome[i] = random.choice(self.alphabet)

        self.calculate_fitness()

    def calculate_fitness(self):
        """Fitness is 2^(number of matching characters)"""
        self.fitness = 1
        for i in range(self.genome_length):
            if self.genome[i] == self.target[i]:
                self.fitness *= 2
```

Listing 4: SEEDS Cell implementing a genetic algorithm to evolve to match a target string. This cell's two parameters, *target* and *mutation*, can be defined in the SentenceCell section of a configuration file.