NEAT Software Doc File

Kenneth O. Stanley
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712 USA
kstanley@cs.utexas.edu

August 16, 2001

1 Introduction

This document is meant to provide enough information to get started using the NEAT neuroevolution software package. It is not intended to be a professional-grade manual. I recommend reading the entire document in order before attempting to use NEAT. Doing so may help you avoid hours of confusion and trial and error.

The software was written to perform experiments based on the NeuroEvolution of Augmenting Topologies method, which is described by Stanley and Miikkulainen (2001). If you don't know about NEAT or about neuroevolution in general, reading the paper will clarify this document. You can find the paper at `http://www.cs.utexas.edu/users/nn/pages/research/nemethods.html#NEAT`. Without reading at least some of that paper, this document probably is of little use to you. The information herein is not intended to be a primer on neuroevolution, as it only explains this particular software package.

Using the system requires becoming familiar with the source code because creating a new experiment requires adding new code to the files `experiments.h` and `experiments.cpp`, which in turn must call evolution methods, mostly in `genetics.cpp`. Thus, we will cover a fair amount of source code in the next few sections.

The next section describes how NEAT is organized into files and what each component of the system does. Section 3 describes different ways that NEAT can be compiled. Section 4 provides a tutorial on how to build and run an experiment using NEAT.

2 System Organization

This section discusses how NEAT is organized, beginning with the different source files used by the system. Most of the source code is heavily commented, so it may be helpful to browse source files while reading this documentation. If you need more information on a method, it should initially be helpful to look at the comments within the source code for that method.

A separate file, `USERDOC`, contains a specific description of every file included in the software distribution. The information below is meant to explain the overall system organization. If you want quick information about a particular file, check `USERDOC`.

NEAT is written in C++. I have compiled NEAT in Linux, under various versions of Debian or Red Hat. With some work, I am sure it can be compiled under other operating systems, but I have not tried to do so (this will be discussed further in Section 3). Following is a list of C++ source files used by NEAT:

- **neat.h** contains the definition for the NEAT class, which most classes in NEAT are derived from. The NEAT class contains static global system parameters and a method for loading them in. It is never actually instantiated as an object.
- **networks.h** and **networks.cpp** contain neural network code for building and activating neural networks as objects that are assemblies of smaller parts (links and nodes). Neural networks are defined in such a way that they can have any topology whatsoever. Some member variables are used to make it easier to generate a neural network from a genome in NEAT. Classes defined include `Link`, `NNode` (Neuron Node), and `Network`.
- **genetics.h** and **genetics.cpp** contain the main NEAT algorithm described by Stanley and Miikkulainen (2001). Genetics covers everything from the gene level to the population level. Classes include `Gene`, `Genome`, `Innovation` (for remembering structural mutations in the current generation), `Species`, `Species_viz` (for holding visualization data), `Organism`, and `Population`. The main generational algorithm is contained in the `epoch` method of the `Population` class.
- **visual.h** and **visual.cpp** contain Gtk−− (a.k.a. gtkmm) based classes for visualization. *You will need gtkmm installed on your system for these files to properly compile*. Gtk stands for "graphics toolkit" and is available under Linux, and possibly other platforms. The "mm" or "−−" mean the object-oriented (C++) version of Gtk (more on on this in section 3). The classes contained here define drawing areas for looking at networks and visualizing speciation. NEAT can display a picture of any evolved network from its genome. It can also show you how speciation looked after a run of evolution.
- **experiments.h** and **experiments.cpp** contain experiments. The main experimental control routines I have included are `xor_test` (evolving xor structure), `pole1_test` (evolving a single pole balancer), and `pole2_test` (evolving double pole balancers, both Markovian and non-Markovian). If you want to add your own experiment, *you should add it to this file*. NEAT allows your code primary control of the "main control loop," so you can design experiments any way you want and call NEAT at your convenience. The experiments I have included are procedures outside of classes. You can write experiments any way you want, but if you want to get started, you may want to follow the format of my experiments. Each has a `test` (the main experiment setup and control), an `epoch` (control code for running a single epoch of trials- note that the genetic part of the epoch is already built into NEAT so you just write the code for doing whatever task it is you want to do and call NEAT to form the next generation), and an `evaluate` function (the code for a single trial of your experiment with a single network). This is my way of dividing up an experiment with NEAT; you don't have to follow it.
- **neatmain.h** and **neatmain.cpp** contain `main`, which seeds the random number generator and launches into Guile, which will be explained next.

NEAT uses Guile as a scripting interface for running experiments from a command line. Information on Guile is available at `http://www.fsf.org/software/guile/guile.html`. If you haven't heard of Guile, it is a library for a version of Scheme (a LISP-like language) that you can compile into your own project. It allows you to make your own C or C++ functions and methods into Scheme functions, and call them from the Scheme command line. Thus, experiments are all callable from Scheme with different parameters.

It turns out that making a C++ method callable from Scheme requires some work. However, a utility called SWIG (`http://www.swig.org/`) allows you to automatically generate wrappers around your C++ code to make it callable from Scheme/Guile. SWIG takes a `.i` file of headers and turns it into C code that plugs into Guile. That is what the file `neatswig.i` is: the file meant for SWIG to process. I used it to generate Guile wrappers for all the functions. SWIG then output `neatswig_wrap.c`, which I copied to `neatswig_wrap.cpp`.

SWIG, at least its current version, isn't perfect, so I had to edit `neatswig_wrap.cpp` a bit to get it to work. If you want to be able to call you experiments from Guile, you may need to do some editing too (I will explain this process in Section 3). For now, just keep in mind that `neatswig_wrap.cpp` contains code to allow you to call experiments and other methods (like visualizations) from a Scheme command line.

If you look at **neatmain.cpp**, you will see that the last thing it does is call Guile using `gh_enter(0, 0, main_prog)`. Thus, when you run NEAT, you begin at the Guile/Scheme command line.

There are several other types of files that are important to NEAT:

- **System Parameter Files (.ne files)** contain all the parameters of neuroevolution (mutation rates, etc..). The suffix -ne stands for NeuroEvolution. When you execute NEAT from the command line, you give it the name of a parameter file. For example:

  ```
  % ./neat pole2_markov.ne
  ```

  activates NEAT with the parameters intended for running the Markovian 2 pole balancing experiment. You can look at parameter files and edit the parameters any way you see fit. Please note that some parameters, namely those with the word "trait," are intended for future expansion of the system and will have no effect on your experiments with the current version of NEAT. Also note that `babies_stolen` is not used in any of the experiments by Stanley and Miikkulainen (2001) and can be left set to 0. (You can e-mail me if you want more info on this option.) Specific information on each parameter can be found in `neat.h`, where each parameter is commented.

- **Genome files** contain descriptions of genomes in NEAT. They can be output by an experiment (for example, to record the champion of a generation), or they can be used for input (for example, as the starter genome for an experiment). They can also be loaded in from Guile to use in tests or printed to the screen as graphs. The following genome files are included with the package: `pole1startgenes, pole2startgenes, pole2startgenes1, pole2startgenes2, and xorstartgenes`. These are loaded in by their respective experiments as the initial spawning genome. All are minimal (zero hidden node) networks. Each genome file contains traits, nodes, and genes. The traits are reserved for future use and can be ignored. The nodes are a list of unique nodes and the "genes" are a list of connections between nodes.

- **Population files** are files containing many genomes. They can be loaded in by the Population class to create a new Population. A Population object can dump itself to a file at any time. You can see how NEAT prints Populations to files by looking at the experiments that are provided. Population files automatically output by NEAT usually begin with the string "`gen_n`" where `n` is the generation when the population was output to a file.

Since a genome file fully specifies a genome from which networks are formed, and upon which all genetic operations take place, it is useful to understand what the numbers in a genome file represent. Here is the genome file `pole2startgenes1`:

```
genomestart 1
trait 1 0.1 0 0 0 0 0 0 0
trait 2 0.2 0 0 0 0 0 0 0
trait 3 0.3 0 0 0 0 0 0 0
node 1 0 1 1
node 2 0 1 1
node 3 0 1 1
node 4 0 1 1
node 5 0 1 1
node 6 0 1 1
node 7 0 1 3
node 8 0 0 2
gene 1 1 8 0.0 0 1 0 1
gene 2 2 8 0.0 0 2 0 1
gene 3 3 8 0.0 0 3 0 1
gene 1 4 8 0.0 0 4 0 1
gene 2 5 8 0.0 0 5 0 1
gene 2 6 8 0.0 0 6 0 1
gene 2 7 8 0.0 0 7 0 1
genomeend 1
```

`traits` can be ignored in this version of NEAT.

`nodes` are followed by the node ID number, the node trait number (which can be ignored), the node type specifier, and the *genetic node label*, which is another type specifier. The reason there are two ways to specify a node type (INPUT, HIDDEN, OUTPUT, or BIAS) is because the `NNode` class is used both inside of networks and inside of genomes. Inside the network, all that matters is whether a node is a sensor or not, but inside the genome, nodes are more strictly classified as one of the four categories mentioned above. This classification scheme could be simplified; it is a legacy that I have not taken the time to clean up.

`genes` are followed by the gene trait number (which has no meaning in current NEAT and can be ignored), the connection in-node, the connection out-node, the weight of the connection, a recur-flag (which is not necessary for the functionality of the system but provides additional information when outputting genomes for user observation), the *innovation number* or historical marking of the gene, another number called a *mutation number* which is always the same as the weight (it is currently a redundant parameter), and finally an enable bit.

3 Compiling

In this section, I will offer some compiling tips, although I do not have tips for any specific platforms.

The Makefile was designed to compile under Debian Linux at my school. You may need to edit the Makefile in order to compile NEAT on your system. Your setup, even under Debian, may have different paths to header or library files, so if you get errors compiling, you should check your paths and fix them to correspond with your system setup.

The most difficult part of compiling NEAT derives from the integration of both Guile and Gtk−− (gtkmm). I highly recommend using both of these utilities if you can. However, if it is too much of a struggle to compile or maintain your experiment with either of these, you can do without them and still use

NEAT's main neuroevolution functions. If you do decide to use them, you will need to make sure they are both installed on your system.

3.1 Compiling with Gtk−−

Gtk−− is used for visualization of both networks and speciation. Most of the Gtk−− based classes are in `visual.h` and `visual.cpp`. If you don't have Gtk−− on your system, you won't be able to visualize using these utilities. Luckily, if you do have Gtk−− installed, it will find its own libraries and dependencies. The main compile line in the Makefile is:

```
neat: networks.o genetics.o visual.o experiments.o neatmain.o neatswig_wrap.o
      \$(CC) \$(CFLAGS) \$(LIBS)
      networks.o genetics.o visual.o experiments.o neatswig_wrap.o neatmain.o
      -o neat `gtkmm-config --cflags --libs`
```

Notice the last part,

```
`gtkmm-config --cflags --libs`
```

This causes gtkmm to append the correct files for inclusion to the compile command, based on your system's paths. So you don't have to go searching for them. However, you still have to be sure you have the right INCLUDE path for gtkmm.

If you decide to do without Gtk−−, you will have to edit the Makefile yourself and remove dependencies on visual.cpp and visual.h. If you find this too difficult, you can e-mail me. I have not spent the time here to create a visual-free version of NEAT. However, doing so should not be difficult. I understand that in other operating systems, perhaps Gtk−− may not be available, in which case, you may want to just write your own visualization routines (and let me know about them!). You may still find my source code a useful reference, at least for developing visualization algorithms.

3.2 Compiling with Guile

Guile is probably the most difficult part of the system to keep integrated as you add new experiments. As explained in section 2, Guile allows you to interact with NEAT from a Scheme command line interface.

Because I realize some users may not find Guile worth the effort (although I highly recommend using it), I have included a Guile-free method of compiling with this distribution. The files `neatmain_ng.h` and `neatmain_ng.cpp` have a different `main` than do the usual `neatmain.h` and `neatmain.cpp`. The letters "`ng`" mean "no Guile." You will find that in the `ng` versions, `main` does not conclude by handing over control to Guile. Instead, if you look inside these files, I have included an example of calling an experiment *directly from main* using hardcoding (as opposed to calling experiments from the Scheme command line as you would with Guile). To compile the Guile-free version of NEAT, call make from the command line:

```
% make neat_ng
```

This call will produce the executable file `neat_ng`, which you can execute with an experimental parameter file just as you would execute `neat`:

```
% ./neat_ng pole2_markov.ne
```

after which the the program will jump immediately into whatever experiment you have called from main.

If you *do* decide to use Guile, which I highly recommend, here is what you need to know. When you add a new experiment by adding a new procedure to `experiments.h` and `experiments.cpp`, you will not automatically be able to call that experiment form the Guile command line. In order to make experiments (or any other procedures) accessible from Guile, you have to also add the same header from `experiments.h` to `neatswig.i`.

For example, the single pole balancing experiment is represented by the following header in `experiments.h`:

```
Population *pole1_test(int gens);
```

The goal is for users to be able to call this from Guile like this:

```
guile> (pole1_test 100)
```

which would begin up to 100 generations of the single pole balancing experiment.

In order to make such calling possible, the header from `experiments.h` was added to `neatswig.i` for processing into Guile by SWIG. Take a look at `neatswig.i` and you will see the very same header, with the word "`extern`" in front of it:

```
extern Population *pole1_test(int gens);
```

Recall that SWIG is used to compile `neatswig.i` into `neatswig_wrap.c`, which will contain a Guile wrapper for any header in `neatswig.i`. Here is how to generate `neatswig_wrap.c` (and then trivially `neatswig_wrap.cpp`):

```
% swig -guile -c++ neatswig.i
% cp neatswig_wrap.c neatswig_wrap.cpp
```

In case you are curious, the wrapper for `pole1_test`, which was generated by SWIG, looks like this (you can also see it for yourself in `neatswig_wrap.cpp`):

```
SCM _wrap_gscm_pole1_test(SCM s_0)
{
         Population * _result;
         int  _arg0;
  char _ptemp[128];
  SCM scmresult;  /* fun1 */
         _arg0 = (int ) gh_scm2long(s_0);
         SCM_DEFER_INTS;
         _result = (Population *)pole1_test(_arg0);
         SCM_ALLOW_INTS;
          SWIG_MakePtr(_ptemp, _result,"_Population_p");
```

```
        scmresult = gh_str02scm(_ptemp);

        return scmresult;
}
```

Of course, you don't have to worry about understanding wrapper code such as above, since it is generated automatically.

One may wonder why I don't just have the call to SWIG in the Makefile. The problem is that I have found that some versions of SWIG on some platforms generate slightly bad code that needs to be manually edited (or edited using a script) to fix the problem. In other words, you will need to slightly edit `neatswig_wrap.cpp` once it has been generated. The error involves the last section of `neatswig_wrap.cpp`, where new Guile procedures are all declared. For example, you can find the place where the procedure for the single pole balancing experiment is declared in `neatswig_wrap.cpp`:

```
gh_new_procedure("pole1_test", (SCM (*)(...)) _wrap_gscm_pole1_test, 1, 0, 0);
```

Notice the part, `(SCM (*)(...))`. I inserted this typecast manually. In fact, SWIG initially generated code with the typecast missing before I edited it:

```
gh_new_procedure("pole1_test", _wrap_gscm_pole1_test, 1, 0, 0);
```

Some compilers require the typecasts and some do not. If yours does, then the raw code output by SWIG will need to have the typecasts inserted manually (all of them are the same: "`(SCM (*)(...))`"). If you look at the file output by `swig -guile -c++ neatswig.i` (when you execute SWIG yourself), you will see that all the `gh_new_procedure` calls are missing the typecast. You will need to insert them yourself by editing `neatswig_wrap.cpp`. When you are done, the file is ready to be integrated into NEAT, and the Makefile will be able to compile it in seamlessly. Note that in the file `neatswig_wrap.cpp` that I provided, you can see the typecasts inserted, but in `neatswig_wrap.c`, which is the raw file output from SWIG, they are missing. Every time you run SWIG, you will have to reinsert them.

Once you have your new experiment added to Guile, you will have the convenience of being able to call it from Guile and do things like capture whatever it returns (i.e. a population) in a Scheme variable for further analysis or testing. Section 4 goes over an entire experiment, from the writing of the control code, to running it from Guile.

3.3 Compiling under different operating systems

NEAT should compile under different operating systems. Hopefully, there are versions of Gtk−−, Guile, and SWIG for your operating system. If not, you may have to do without them (and forego visualizations and the scripting interface). However, most of the code should be operating-system independent. Of course, many paths in the Makefile may need to be edited.

4 Tutorial

In this section I will go over the steps for creating a new experiment from scratch. The experiment I will use as an example is the double pole balancing experiment. Double pole balancing is already included in this distribution. The experiment is called `pole2_test` in `experiments.h`. We will pretend it does not yet exist for the sake of the tutorial.

4.1 Creating Headers and Adding to Guile

The first goal is to declare the procedures that will contain the experiment. We do this in `experiments.h`:

```
Population *pole2_test(int gens,int velocity);
bool pole2_evaluate(Organism *org,bool velocity,CartPole *thecart);
int pole2_epoch(Population *pop,int generation,char *filename,
                bool velocity, CartPole *thecart,int &champgenes,
                int &champnodes, int &winnernum, ofstream &oFile);
```

Notice there are three main procedures: the test, the evaluation, and the epoch. I divide up experiments in this way, but of course you don't have to. The test, which is the main experiment control routine, takes an argument for the number of generations, and whether or not velocity inputs will be made available to the network. We declare velocity as an integer so that it can be interpreted by SWIG (Section 3.2) , which does not interpret booleans. `pole2_test` will set up a new a population and run a series of epochs by calling `pole2_epoch`, which in turn will call `pole2_evaluation` for every individual network attempt. `pole2_epoch` calls the `epoch` method of the `Population` class in order to run the NEAT algorithm for each generation.

In order for Guile to be able to call `pole2_test`, we have to add the header to `neatswig.i` as explained in section 3.2. So we add the header:

```
extern Population *pole2_test(int gens,int velocity);
```

to `neatswig.i`, run this updated file through SWIG, and get the new wrapper file `neatswig_wrap.c`. We then copy `neatswig_wrap.c` to `neatswig_wrap.cpp`. Finally, we edit `neatswig_wrap.cpp` in order to insert the typecasts that SWIG doesn't put in. (All of these steps are explained in detail in section 3.2.) Our experiment is now callable from Guile like this:

```
guile> (pole2_test 100 0)
```

which means, "run the double pole balancing experiment for 100 generations without providing velocity information."

4.2 The main epoch loop

The main job of `pole2_test` is to run an experiment for `gens` generations and output some statistics about what happened. `pole2_test` should also be able to run more than one experiment and average the statistics over all the runs.

An experiment must begin by creating a population. A new population is spawned off of a *starter genome*. The starter genome is a file containing the *structure* that all the first-generation genomes will contain. Following NEAT tradition, starter genomes should be minimal structures, which NEAT will grow over the coming generations. `pole2_test` has two possible starting configurations, depending on whether or not the network will be receiving velocity information (i.e. whether or not the experiment is Markovian). The two starter genomes look as follows. The file `pole2_startgenes1` is the starter genome for the case where velocity is provided[1]:

[1] Recall that the `trait` lines can be ignored

```
genomestart 1
trait 1 0.1 0 0 0 0 0 0 0
trait 2 0.2 0 0 0 0 0 0 0
trait 3 0.3 0 0 0 0 0 0 0
node 1 0 1 1
node 2 0 1 1
node 3 0 1 1
node 4 0 1 1
node 5 0 1 1
node 6 0 1 1
node 7 0 1 3
node 8 0 0 2
gene 1 1 8 0.0 0 1 0 1
gene 2 2 8 0.0 0 2 0 1
gene 3 3 8 0.0 0 3 0 1
gene 1 4 8 0.0 0 4 0 1
gene 2 5 8 0.0 0 5 0 1
gene 2 6 8 0.0 0 6 0 1
gene 2 7 8 0.0 0 7 0 1
genomeend 1
```

IMPORTANT NOTE: In order to create starter genomes, you can write them to a text file just as I did. Please note that you must have *at least one trait* in your genome even though traits are not used by the system currently. The trait(s) you include will have no effect on evolution, but without at least one the system will crash. Traits are reserved for future system explansion. The traits go at the top of the genome file as shown above, starting with a *trait number* (the first trait should be number 1), following by 8 parameters, the values of which do not matter.

The file `pole2_startgenes2` is for the case where velocity inputs are not provided, and therefore has fewer genes:

```
genomestart 1
trait 1 0.1 0 0 0 0 0 0 0
node 1 0 1 1
node 2 0 1 1
node 3 0 1 1
node 4 0 1 3
node 5 0 0 2
gene 1 1 5 0.0 0 1 0 1
gene 1 2 5 0.0 0 2 0 1
gene 1 3 5 0.0 0 3 0 1
gene 1 4 5 0.0 0 4 0 1
genomeend 1
```

The way we load in these starter genomes into an experiment is by calling the `new Genome` constructor off of the file `pole2startgenes` (*without a number following it!*):

```
ifstream iFile("pole2startgenes",ios::in);
```

```
...

cout<<"Reading in the start genome"<<endl;
//Read in the start Genome
iFile>>curword; //Pass the token ``genomestart''
iFile>>id;      //Load in the genome id number
cout<<"Reading in Genome id "<<id<<endl;
start_genome=new Genome(id,iFile);  // <-- Here is the constructor
iFile.close();
```

The question then comes up, what is in `pole2_startgenes`? The answer is, in my method for running the experiment, I copy either `pole2_startgenes1` or `pole2_startgenes2` into `pole2_startgenes`, depending on whether I plan to run the experiment with or without velocity inputs. Of course, it could have been done differently, where the experiment automatically decides which file to load in. However, I left it open ended so that I could start with *other* configurations as well, such as networks with some structure already built in.

In order to spawn a population off of the newly loaded starter genome, we use a special population constructor:

```
//Spawn the Population from starter gene
cout<<"Spawning Population off Genome"<<endl;
pop=new Population(start_genome,NEAT::pop_size);
```

In order to allow multiple experiments (with each run `gens` generations long), the global system parameter `NEAT::num_runs`, which is input from `.ne` parameter files, is used:

```
for (run=0;run<NEAT::num_runs;run++) {
```

Note that inside each run, a new population is spawned, while the previous population was deleted at the end of the run.

Inside of each run, some experiment-specific code is run to set up the pole and cart system:

```
//Create the Cart
thecart=new CartPole(true,velocity);
```

This setup is followed by the main generation loop. Each generation calls the epoch procedure:

```
highscore=pole2_epoch(pop,gen,fnamebuf->str(),velocity, thecart,
                      champg, champn,winnernum,oFile);
```

Many of the arguments are used for statistics collection. If you look at the generation loop in `experiments.cpp`, you can see the altered variables being put back into large stat-collection arrays. There is also a stopping case, which is useful for experiments like this where the system eventually finds a "winner":

```
if (((pop->winnergen)!=0)&&(gen==(pop->winnergen))) {
        /* Calculate the number of evaluations it took
           to find a winner */
        winnergens[run]=NEAT::pop_size*(gen-1)+winnernum;
        gen=gens+1;
}
```

After the main generation loop, stats are dumped out of arrays into the file `statout` (`xor_test` and `pole1_test` on the other hand only print stats to screen). At the very end of the procedure, the population `pop` is returned. Returning the population allows Guile to "catch" the final population so that it can be manipulated or analyzed further.

4.3 Single Epoch Control

The procedure `pole2_epoch` runs individual epochs in this experiment. The main goal of a single epoch is to evaluate every member of the population on some test, and then reproduce to create the next generation using the NEAT method.

The main loop cycles through the organisms in the population, and sends each one to a test. The `Organism` class is used to represent population members. Each organism contains a network in addition to statistical information like fitness (You can look at the `Organism` class in `genetics.h`). The organisms are initially generated from genomes that NEAT automatically translates into the networks inside the organisms.

Each organism plays the double pole balancing task, where they receive their fitnesses:

```
if (pole2_evaluate((*curorg),velocity,thecart)) win=true;
```

After the task loop, some statistics are collected and a "snapshot" is taken of the population:

```
  if ((generation%1)==0)
    pop->snapshot();
```

The modulus allows us to control how frequently snapshots are taken. The snapshot is what allows visualization of the run after completion.

After the snapshot is taken, the procedure diverges depending on whether we are playing the Markovian or non-Markovian version of the problem. In the Markovian case, some stats are taken. In contrast, the non-Markovian case concludes with a complicated sequence of tests administered only to the population champion. These tests are meant to determine if the champion is a "winner," according to special criteria (in the non-Markovian case) described by Stanley and Miikkulainen (2001). The criteria were first introduced by Gruau et al. (1996).

For the purposes of this tutorial, the important point is that it is possible to do whatever tests you want to your population in whatever order you require. If you need the champion at the end of an epoch for further testing, as we do here, you can do something similar to what I do in `pole2_epoch`. In fact, in this procedure, it is not exactly the champion that is chosen for the winner test. Rather, the chosen organism is actually the champion of the highest best-performing species that has not been tested since it last improved. Thus, we don't bother testing champions from species that were checked in previous generations without

improving since. In any case, the main point is that you don't just have to test every organism once, and you can run different tests on different organisms.

At the very end of the procedure, we print the entire population to a file (depending on the global parameter `NEAT::print_every`) to decide if this generation should be output):

```
//Only print to file every print_every generations
if  (win||
     ((generation%(NEAT::print_every))==0))
  pop->print_to_file_by_species(filename);
```

We also print the champ genome to a file:

```
print_Genome_tofile(champ->gnome,"champ");
```

Finally, reproduction and the creation of the next generation take place through NEAT:

```
pop->epoch(generation);
```

Notice that NEAT is called from the experiment rather than vice versa. This organization means that the experiment controls the main loop in the program, so you do not have to worry about how to plug your experiment into NEAT.

4.4 Single Network Evaluation

`pole2_evaluate` is the procedure used to evaluate a single network on the pole balancing task. We begin by extracting the network from its respective organism:

```
net=org->net;
```

The fitness is then determined by actually playing the network at the task using:[2]

```
org->fitness = thecart->evalNet(net,thresh);
```

`evalNet` is the method for running a cart-pole physical system. I will return to it in a moment. The evaluation is followed by some informational screen output, which you may or may not want to have:

```
#ifndef NO_SCREEN_OUT
  if (org->pop_champ_child)
    cout<<" <<DUPLICATE OF CHAMPION>> ";

  //Output to screen
  cout<<"Org "<<(org->gnome)->genome_id<<" fitness: "<<org->fitness;
  cout<<" ("<<(org->gnome)->genes.size();
```

[2]It should be noted that the `thresh` variable, which I once used for debugging (making sure there was not an infinite loop), is now obsolete and can be ignored.

```
  cout<<" / "<<(org->gnome)->nodes.size()<<")";
  cout<<"   ";
  if (org->mut_struct_baby) cout<<" [struct]";
  if (org->mate_baby) cout<<" [mate]";
  cout<<endl;
#endif
```

Notice I used the compiler variable `NO_SCREEN_OUT` to control output. At the very top of `experiments.cpp`, you can either `#define` this variable or not, depending on whether or not you want text output to the screen during an experiment.

The procedure concludes with several branches depending on which experiment, and which part of which experiment, we are currently performing. It returns a flag if it detects a winner.

We should look at `evalNet`, where the actual activation gets run through the network. The physics code for this system was written by Faustino Gomez.

`evalNet` has two branches, one for Markov, and one for non-Markov. Let us examine the Markov case. Activation is performed in a loop, beginning with loading the sensors:

```
    while (steps++ < maxFitness) {


      input[0] = state[0] / 4.8;
      input[1] = state[1] /2;
      input[2] = state[2]  / 0.52;
      input[3] = state[3] /2;
      input[4] = state[4] / 0.52;
      input[5] = state[5] /2;
      input[6] = .5;

      net->load_sensors(input);
               .
               .
               .
```

You can see that networks have a method for loading their sensors off of an array. Of course, the network will only load in as many sensors as it has, so if you have an array of more numbers than sensors in the network, the leftovers will be truncated. If your array is too small, you can end up with a memory leak. The system assumes the array is the same size as the number of sensors (the sensors include a bias, which is a sensor always set to the same number).

After loading sensors, the network is activated:

```
if (!(net->activate())) return 1.0;
```

The conditional is checking for an error; generally the network should always activate when requested. The above statement activates the network for a single time step.

After the network is activated, we need to extract its output:

```
output=(*(net->outputs.begin()))->activation;
```

The network used in this experiment has a single output. You may have networks with multiple outputs. Outputs are stored in the public `outputs` member variable of the `Network` class in a Standard Template Library list. So you can cycle through all your outputs as you move through the list in order to extract activations. The `Network` class is in `networks.h`. If you do not know about STL lists, you can read up about them in most references on C++.

The extracted output is used to perform the next action of the cart system:

```
performAction(output);

if (outsideBounds())    // if failure
    break;                      // stop it now
```

The last check exits the activation loop if the pole falls down too low. The fitness is set to the number of steps the pole stayed above the bounds:

```
return (double) steps;
```

The main point is that you have to load in the inputs, which presumably come from whatever model you are running, then activate the net, then extract the outputs, and feed them back into the model. All of this activity typically happens in a loop where a network is activated off the current sensors on each iteration of the loop (as you can observe in `CartPole::evalNet`).

However, there is a kind of task where you will want to do things differently, so I will go over it here, even though it is not related to the pole balancing task. In classification tasks such as XOR, you don't want to run activation in a continuous interactive loop, with the inputs and outputs always changing. In classification, you just want to run the inputs through the network to the outputs *once*, and then extract the answer. Here is how this is done in XOR:

```
net->load_sensors(in[count]);

//Relax net and get output
//use depth to ensure relaxation
for (relax=0;relax<=net_depth;relax++) {
  success=net->activate();
  this_out=(*(net->outputs.begin()))->activation;
}

out[count]=(*(net->outputs.begin()))->activation;

net->flush();
```

The key concept is *relaxation*. Because NEAT evolves networks of arbitrary topologies, there can be many hidden layers. A single time step only allows for activation to propagate from one hidden layer to the next. Thus, we need as many time steps as there are layers in the net. In other words, we use the *depth* of the network to ensure relaxation. The variable `net_depth`, was set using:

```
net_depth=net->max_depth();
```

max_depth is a Network class method that returns the depth of the network. Thus we loop on activating the network for as many steps as the network is deep, at which point we know the network is relaxed.

It is also worth noting that in Markovian classification tasks, there is usually no reason to evolve recurrent links. You can make it impossible for NEAT to evolve recurrencies by setting recur_only_prob to 0 in the parameter (.ne) file of your experiment.

4.5 Running the experiment

With the experiment code complete, we can now run the experiment from Guile.[3] After executing make, a new file, neat, contains NEAT's executable. NEAT is run from the command line with an appropriate parameter (.ne) file. Let us say we are going to run Markovian pole balancing. First, we make sure the correct (Markovian) starter genome is in pole2startgenes:

```
% cp pole2startgenes1 pole2startgenes
```

Then we can run NEAT:

```
% ./neat pole2_markov.ne
```

Upon execution, NEAT confirms the parameters in the .ne file and then jumps directly into Guile, giving you the Guile command prompt:

```
guile>
```

From this prompt, you can call any function from neatswig.i. As a courtesy, SWIG outputs a summary of such functions in neatswig_wrap.doc. In our case, we are interested in running pole2_test:

```
guile> (pole2_test 100 1)
```

This command tells Guile to run the pole2_test procedure for 100 generations, in the Markovian mode (with velocity inputs provided). However, an even better way to run the experiment is to catch the final population in a variable for further analysis. We can do this because pole2_test returns the population at the end of the experiment:

```
guile> (define p (pole2_test 100 1))
```

Now the Scheme variable p will contain the final population after the experiment is complete.

During the experiment, fitness information is output during each generation, depending on the setting of NO_SCREEN_OUT. After each generation, a summary of species is output with their highest fitnesses and data on when it last improved. Populations are dumped every print_every (a system parameter)

[3] If you chose not to use Guile, then you have to hardcode the calls in this section into main as C++ calls to your experiment

generations into files named gen_n where n is the generation number of the population. These files can be viewed by any text editor. They have comments containing fitnesses and species.

Each run will terminate every time a winner is found. In the Markovian case, a winner is a network that balances a pole for 100,000 time steps. When a winner is found, the entire population will be output to a gen file regardless of the setting of print_every. If you look inside this file, search for the text WINNER and you will find the winning genome (it is commented with the word WINNER).

NEAT will run num_runs experiments and terminate with a summary of the session. Statistics over all runs are printed both to the screen and to a file called statout. The statistics include the number of nodes and genes in champions as well as number of evaluations to completion. The original call to pole2_test returns the population in its final state from the final run. In our example, the Guile variable p receives the population.

4.6 Visualizations

Once we have the population p, we can visualize the speciation that took place over the run, as long as we have Gtk−− installed on the computer. visualize is a Population class method:

```
void Population::visualize(int width,int height,int start_gen,
                           int stop_gen)
```

Since the Scheme variable p contains a population object, we can call its visualize method from Guile. Suppose the run lasted 22 generations. Then we can visualize the whole thing like this:

```
guile> (Population_visualize p 500 700 1 22)
```

Visualizations of speciation display species expanding, contracting, appearing, and disappearing over generations. In addition, colors indicate rising fitness. You can visualize any generation interval that falls between 1 and the final generation of the run. For more information, refer to Stanley and Miikkulainen (2001), where visualizations are explain in-depth.

In addition to getting a visualization of an entire run, we also want to be able to see pictures of arbitrary networks that have evolved. Networks can be visualized off of their respective genomes or genome files. In every generation of the double pole balancing experiment, NEAT outputs the champ genome from that generation to the file champ (so it is always being overwritten). Here is how NEAT did this from experiments.cpp:

```
print_Genome_tofile(champ->gnome,"champ");
```

Thus, at the end of a run, we can easily visualize the final champ from Guile, since its genome is sitting in the file champ:

```
guile> (display_Genome "champ" 700 700)
```

A 700 by 700 pixel window pops up with a picture of the champ . Blue connections are inhibitory; black are excitatory. Nodes id's appear next to their respective nodes. Note that node id's represent a chronology of the appearance of each node over the course of evolution. So if we see the number 50, we know it was the 50th node to appear through a mutation in the history of the run. Recurrent links are represented as loops. For nonrecurrent links, line thickness is proportional to strength of connection.

4.7 Your own experiment

When you write your experiment, you can borrow heavily from the code already in `experiments.cpp`. Most of the work will involve writing the task simulation, which was a cart and pole physics model in this tutorial. The evolutionary control structure is mostly already written for you. Of course, you may want to make changes for your purposes. The `pole2_test` is the most complicated experiment provided. If you are doing something simpler, you may want to borrow more from `pole1_test` or `xor_test`, which are easier to follow.

It is possible you will want to do something radically different. The only thing you really need to do in any experiment is call `epoch` when you are ready to produce the next generation based on the fitnesses of the current ones. Anything else is conceivable. You can even have multiple populations (as in some competitive coevolution experiments). In some cases, you may need to change something about how NEAT itself works and edit a file other than `experiments.cpp` or `experiments.h`. For example, you may want to change how stagnation is detected or some other subtle aspects of evolution itself inside of `genetics.cpp` or `genetics.h`. Of course, such changes are possible and could be important. However, they would require familiarizing yourself with the genetic control code from NEAT, which may take some time.

For simple experiments, however, such tinkering is unnecessary, and NEAT should suffice as is.

5 Conclusion

I hope that this software will be a useful starting point for your own explorations in neuroevolution. The software is provided as is. However, I will do my best to maintain it and accommodate suggestions. If you want to be notified of future releases of the software or have questions, comments, bug reports or suggestions, send email to `kstanley@cs.utexas.edu`. In addition, I would be happy to hear any applications you have implemented or are considering for NEAT.

Acknowledgments

The research that produced the NEAT software was supported in part by the National Science Foundation under grant IIS-0083776. Thanks to Faustino Gomez for providing pole balancing code.

References

Gruau, F., Whitley, D., and Pyeatt, L. (1996). A comparison between cellular encoding and direct encoding for genetic neural networks. In Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, 81–89. Cambridge, MA: MIT Press.

Stanley, K., and Miikkulainen, R. (2001). Evolving neural networks through augmenting topologies. Technical Report AI2001-290, Department of Computer Sciences, The University of Texas at Austin.