SapioGo
Design Document

Ryan Flannery
squid@fuse.net

Kevin Upchurch
upchurks@email.uc.edu

Advisor: John Schlipf
john.schlipf@uc.edu

June 8, 2004

1 Design

1.1 Go Text Protocol (GTP) Library

The GTP library requires at least version 3.4 of GNU Go in order to use this class properly. Calling this constructor it will spawn an instance of GNU Go and uses a pipe to send and receive GTP commands and responses. When the GTP object is destroyed, or the `kill()` command is called, the GNU Go instance is terminated.

1.1.1 Class `Gtp`

The GTP library also includes a way to talk Go Modem Protocol allowing itself to be used with programs that only talk in GMP, such as cgoban. Calling the `Connect()` command the use is able to use GMP as well as take advantage of Gtp. This is because the Connect command also flags the bool `gmp`, so that when you 'pass' or make a move with GTP you will also send a corresponding GMP request.

Some other features of the GTP library is the `tell_gnugo` command and the `ask_gnugo` command, both of which are private members. These two functions are the basis in which the GTP library is founded. The `tell_gnugo` command actually sends a GTP command to GNU Go. The `ask_gnugo` reads from the return pipe and puts the return pipe in to a private character array called `gnugo_line[]`. The meat of the GTP class is in the `MakeGtpMove(int x, int y, int iswhite)` which takes care of moving pieces on the board, as well as incrementing `numbermoves`. `Islegal(int x,int y,int iswhite)` is similar except that it returns whether the X and Y coordinate of the board is a legal

move or not. The `newscore(int isWhite)` is used to calculate the score at the end of the game. If the board is mostly empty then this function may take longer than normal because it is harder to calculate what territory belongs to whom.

1.2 Go Modem Protocol (GMP) Library

Our implementation of the `GoGMP` class was modified from code originally done in C by William Shubert, but was not written in an Object Oriented Model. GMP is able to set up a new game, and send moves or passes. You can also query player information with the calls to functions such as `gmp_komi(), gmp_size(), gmp_handicap(), gmp_chineseRules()`, and `gmp_iAmWhite()`. `Gmp_startGame(int size, int handicap, float komi, int chineseRules, int iAmWhite)` sets up the board for a soon to beginning game. Once `gmp_startGame` is called you would then call the `waitForNewGame()` which waits for the other client to initialize their game. `gmp_sendMove(int x, int y)` is used to send moves while `gmp_sendPass()` is used to send passes.

1.3 Neural Network Library

The Neural Network library consists of two ADT's, `Neurode` and `Network`. A `Network` object contains a set of `Neurode` objects (its *neuron population*). As such, the `Neurode` class is never meant to be used by the programmer using the Neural Network library. All interaction with neurodes is encapsulated in the `Network` class.

1.3.1 Class `Neurode`

The `Neurode` ADT encapsulates all of the properties of an artificial neuron. It has two public data members, `InputWeights` and `OutputWeights`, which are pointers to arrays of floating point values. These two arrays correspond to the input and output weight vectors, respectively, associated with each neuron.

There are two other public data members, `InputWeightSize` and `OutputWeightSize`, which correspond to the size of the two input weight vectors.

The default constructor, `Neurode()`, creates an empty neurode with no weight vectors, and is never intended to be used. Instead, there is a constructor `Neurode(int InputSize, int OutputSize)` which creates a neurode with the given dimensions. The input and output vectors are sized according to the `InputSize` and `OutputSize` parameters, respectively. All memory is dynamically allocated.

1.3.2 Class `Network`

The `Network` ADT is the class that a programmer would interact with when using the Neural Network Library. Class `Network` encapsulates all of the properties and methods associated with a two-layer, feed-forward, artificial neural network.

Types `InputVector` *and* `OutputVector`. Class `Network` includes the definition of two other ADT's, `InputVector` and `OutputVector`. Both are simply `typedef`'s of the Standard Template Library's `vector` class, of type float (`vector<float>`).

Creating a `Network`. To create a `Network`, the dimensions of the neural network must be specified. These include three parameters: the input size, the hidden size, and the output size. The corresponding constructor is:

```
Network(unsigned int InputLayerSize, unsigned int
  HiddenLayerSize, unsigned int OutputLayerSize)
```

Randomizing and Clearing a `Network`. Once a `Network` has been created, all of the weights associated with each `Neurode` can be set to random floating-point values between 0 and a provided upper-bound. This is done with the `Network`'s public method, `void randomize(float UpperBound)`.

A `Network`'s structure can also be completely cleared, using the public method `void clear(void)`. All `Neurodes` are removed and the dimensions of the `Network` are set to 0.

Iterating a `Network`. Feeding an input vector into the network and producing an output vector is called *iterating* the network. Iterating is accomplished using the following public method:

```
void iterate(InputVector &input, OutputVector &output)
```

The `input` vector, although passed by reference for memory considerations, remains unchanged. The `output` vector is cleared and then re-filled with the output of the neural network.

Storing/Loading a `Network` *to/from a File.* Class `Network` also has the ability to store its structure to a file using the `bool save(char *filename)` public method. The network is saved in a simple ASCII text format with one weight (a `float` value) stored per-line. The `save()` method returns `true` if it finished successfully and `false` otherwise.

A `Network` can then be loaded from file using the `bool load(char *filename)` public method. Any existing network structure is cleared before the network is loaded. Similar to the `save()` method, `load()` returns `true` if it finished successfully and `false` otherwise.

There is even a constructor, `Network(char *filename)`, that will create a `Network` loaded from the provided file.

1.4 Sane-Network Library

The Sane-Network library consists of a single ADT, class `SaneNetwork`, which adds all the functionality of the SANE algorithm to the `Network` ADT, including selecting a random subset (blueprint) of the neurode population, iterating with a given blueprint, rating the performance of a blueprint, and breeding an entire network.

1.4.1 Class `SaneNetwork`

`SaneNetwork` is a child-class of the `Network` ADT described above. The descriptions below outline the design of the `SaneNetwork` class and how to use it.

Creating a `SaneNetwork`. To create a `SaneNetwork`, the dimensions of the underlying neural network must be specified just as for the `Network` class, along with additional parameters required for the SANE algorithm. These include three parameters: the mutation rate, the size of the blueprints selected, and the minimum number of times every neurode must be used in a blueprint before the network may be breed. The corresponding constructor is:

```
SaneNetwork(unsigned int InputLayerSize, unsigned int
HiddenLayerSize, unsigned int OutputLayerSize, unsigned int
MuatationRate, unsigned int BlueprintSize, unsigned int
MinimumUsage)
```

Selecting a Random Blueprint. To select a random blueprint from the `SaneNetwork`, there is a public method that returns a randomized blueprint. Its definition is:

```
blueprint randomizeBlueprint(void);
```

Iterating with a Blueprint. Once a blueprint has been selected, the `SaneNetwork` may be iterated just as a `Network` would, using the following method:

```
void iterate(InputVector &input, OutputVector &output, blueprint
                        *bprint = NULL);
```

Note that the third parameter is a pointer to a blueprint. If no blueprint is provided, the entire network is iterated. Otherwise, only those neurodes in the passed blueprint are used in the iteration.

Rating the Performance of a Blueprint. After a blueprint has been used for a given application, its performance may be rated using a floating-point value and the following public method of `SaneNetwork`:

```
void rate(float rating, blueprint *bprint = NULL);
```

Breeding a `SaneNetwork`. Once all of the neurodes in a `SaneNetwork` have been used in enough applications, the neurodes may be breed using the following public method of `SaneNetwork`: `void breed(void);`

1.5 SapioGo

The design of SapioGo is broken into two client programs: `sapiogo` and `sapiogo-coevo`. The first program is designed to play against GNU Go and human opponents, while the second program is designed for coevolution, where SapioGo plays against itself.

1.5.1 Program `sapiogo`

The `sapiogo` client program is relatively simple, since the `GTP` and `SaneNetwork` libraries encapsulate all aspects of playing Go and learning, respectively. The `sapiogo` program works by either creating a new, random network or loading in an existing network from a file. For each generation, random blueprints are selected until all neurodes in the network have been used the minimum number of times (a command-line parameter). For each blueprint that is selected, a game of Go played using only the neurodes in that blueprint. Afterwards, the neurodes of the blueprint are rated with the score of the game. After all of the neurodes have been used the minimum number of times, the neurodes are breed and a new generation begins. Figure 1.5 complete is pseudo-code for the `sapiogo` program.

```
1  Create a random network or load-in a network from a file
2  For X generations...
3      Create a new SaneNetwork
4      Until all neurodes in the network have been used Y number of times...
5          Select a random blueprint from the SaneNetwork
6          Play a game of Go using only the blueprint selected
7          Rate the neurodes of the blueprint with the score of the game
8      Breed the neurodes of the SaneNetwork
```

Figure 1: `sapiogo` Pseudo Code.

1.5.2 Program `sapiogo-coevo`

The `sapiogo-coevo` client program is, like the `sapiogo` program, also relatively simple. The `sapiogo-coevo` program starts by creating either a new, random network or loading in an existing network from a file. For each generation, two random blueprints are selected at a time, one called 'white' and the other called 'black'. A game of Go is played pitting the two blueprints against each other. After the game is finished, the neurodes of each blueprint are rated with the score that blueprint received. This continues until all neurodes have been used the minimum number of times. Afterwards, the neurodes are breed and a new generation begins. Figure 1.6 is complete pseudo-code for the `sapiogo-coevo` program.

```
1  Create a random network or load-in a network from a file
2  For X generations...
3      Create a new SaneNetwork
4      Until all neurodes in the network have been used Y number of times...
5          Select a random blueprint from the SaneNetwork, call it 'white'
6          Select a random blueprint from the SaneNetwork, call it 'black'
7          Play a game of Go between the white and black blueprints
8          Rate the white blueprint with its score
9          Rate the black blueprint with its score
10     Breed the neurodes of the SaneNetwork
```

Figure 2: `sapiogo-coevo` Pseudo Code.