**NEAT Mario**

System Manual

Christopher Bruns, Jack Van Gent, Jacob Espenscheid

**Abstract**

Learning to play Super Mario World with genetic algorithms and a neural network. A detailed operations and programming guide of the structure of the system.

May 2, 2016

Dr. Ben Schafer: CS 4620/5620, Intelligent Systems

University of Northern Iowa Computer Science department

Table of Contents

Operations Guide 3

System Requirements (Christopher Bruns) 4

Installation (Jacob Espenscheid) 7

Modifying the System (Christopher Bruns) 8

System Backups (Christopher Bruns) 11

Failure Recovery (Jacob Espenscheid) 12

Programming Guide 13

Software Structure Chart (Jack Van Gent) 14

Data Descriptions (Jack Van Gent) 15

Source Code Catalog (Jack Van Gent) 16

Test Data (Jacob Espenscheid) 18

# Operations Guide

## System Requirements

For the system to run you will need the following requirements: a Windows based computer and the CD-ROM that contains the BizHawk emulator, the BWDHmaster.lua file, and the ROM for the game.

The lua file can be opened and edited in a variety of environments. For our purposes we used LuaDevelopmentTools which is built on Eclipse, Notepad ++, and Atom. However other code editors will be able to work to make changes to the working code and can still be implemented with BizHawk.

BizHawk works on Windows based computers and is downloaded from the internet. The emulator works with minimal system requirements as long as the it is Windows based. There are alternate ways to compile and work on Linux and MacOS devices however our group was unable to get them to compile without errors and performance issues so to ensure optimal performance a Windows computer will be the best solution. To compile and build BizHawk follow these steps as outlined from the TAS website (<http://tasvideos.org/Bizhawk/Compiling.html>). However using the CD-ROM submitted you will be able to bypass these steps, they have been included for referencing and for the event that they are needed for troubleshooting purposes.

1. Have a compatible Visual Studio on your device, the site recommends Visual Studio Community 2013 which can be found here, <https://www.visualstudio.com/en-us/products/visual-studio-community-vs>.
2. Install TortoiseSVN, found here <https://tortoisesvn.net/>. This is a Windows shell extension
3. Create a directory and make sure it's empty. It's always a good idea to pick a directory name without spaces in it; this tutorial will use "C:\BizHawk".
4. Right-click in it and pick "SVN Checkout". In the box "URL of repository" enter "<https://github.com/TASVideos/bizhawk>". Leave everything else blank and click "OK". Now some files will be downloaded from GitHub.
5. Find "BizHawk.sln" among the downloaded files and double-click it. If you get prompted to sign in, pick "not now" unless you want your life more complicated on account of someone trying too hard to make it too simple. If this is the first time you've run Visual Studio, you'll get prompted for "Development Settings". The current setting is "General". Change it to "Visual C++". If you choose anything else, nobody will be able to tell give you suggestions for hotkeys. Then pick "Start Visual Studio".
6. You should have the \*.sln file open now and see a "Solution Explorer". Look for "BizHawk.Client.EmuHawk", right-click it, and pick "Set As Startup Project". Now go to "Build | Configuration Manager" in the menu, change the active solution configuration to "Release" and pick "Close".
7. Finally you're at the point where you'll usually be at when checking out a new BizHawk build: hit Ctrl+F5 to build and run it.

Before you can use some of the emulator cores, you'll need to drop in some firmware files; the right spot to put them will be at "C:\BizHawk\output\Firmware".

Now, for an important part, to pull down SVN updates so you can build them, right-click "C:\BizHawk" and pick "SVN Update". You can now repeat the build-and-run step in Visual Studio to test the new build.

If your SVN ever gets wrecked and can't build then close Visual Studio, make a backup of your "C:\BizHawk" directory, then make a new one and go back to step 3.

## Installation

This system is only able to be installed on windows computers.

**Step 1: Run bizhawk\_prereqs.exe**

This file will install these to make sure you have the requirements to run BizHawk. Here is a list of everything that is downloaded on our version of bizhawk\_prereqs.

Windows Imaging Component (.net 4.0 prerequisite for older OS)

Visual C++ 2010 SP1 Runtime

Visual C++ 2015 Runtime

.Net Framework v4.0.30319

DirectX Web Update

**Step 2: Run EmuHawk.exe**

To get to this file from the root directory open the BizHawk-1.11.4 folder. EmuHawk.exe is located in this file. If everything went correctly then this should open BizHawk.

## Modifying the System

Within our code we use a neural network trained by a genetic algorithm and different types of mutations. There are a variety of different combinations that can be applied within. The ways to alter these are outlined in this section with line numbers within the code provided for easy accessibility.

Global Variable Alterations:

Our global variables are all located at the top of the code. These variables set the base values for mutation chances, population, and neuron information. The majority of tests ran originally involved a population of 100, however this can be changed and modified. We recommend a size of at least 50 and no more than 300 (line 14).

Mutation chances are altered and updated in functions within but the global variables may be changed in the globals as well (lines 2-10). The values would be best to be kept less than 2, however if they are set higher it will adjust within the code and stabilize later as the program runs.

Altering Mutations:

    Options for different variations you can choose from are listed in this paragraph and then more detailed instructions are in the paragraphs following. Node mutations: nodeMutation, tailNodeMutation, and twoNodeMutation. Crossover mutations: crossoverMutation, onePointCrossover, and twoPointCrossover. Link mutations: linkMutation and doubleLinkMutation.

To determine the mutations you wish to perform there are a few ways to do this. The easiest way will be to pick one of the three mutate functions and one of the two breed functions. However there are other ways to more personalize the code, which will also be outlined.

Within the newGeneration() function new chromosomes are added to the species through breeding and crossovers. On line 1362 and line 1368 you may alter the call for which type of function you wish between altBreedChild(species) and breedChild(species). We recommend keeping both the same. The altBreedChild function (line 1174) will randomly choose between three different crossover techniques throughout the time it is ran. While the breedChild function (line 1150) only has one crossover technique implemented but this may be changed. On line 1156 you may change crossoverMutation to onePointCrossover or twoPointCrossover instead if you choose to. Keep two chromosomes as the parameters as it is already established.

To determine the types of mutations the easiest way to change this is within the breedChild() and altBreedChild() functions where the mutate function is called. On line 1162 and 1199 of these functions the child created is mutated. The function options are mutateChromosome, randomMutateChromosome, and generationMutateChromsome.

mutateChromosome allows for the fewest variations but you are able to determine which ones you want to perform. On line 855 determine the type of node mutation you would like to take place from the options listed earlier, again these were nodeMutation, twoNodeMutation, and tailNodeMutation. On 834 and 845 you may changes these to both linkMutations or doubleLinkMutations.

The other two functions, the generationMutateChromosome() and randomMutateChromosome() take care of picking between these options on their own. The generation option picks from the options based on the current generation number and also allows for more mutations to take place in earlier generations. Random mutate is going to allow for a random choice between all of the options. While altering these is not necessary you may do so if you wish. The random function begins on line 880 and the generation function starts on 958.

## System Backups

Our system will backup its generations automatically while the program is running. These are able to be reloaded back into the system to continue where it left off if desired. Lines 23, 25, 27, 29, and 32 are set up with comments to allow you to enter the description of the run and the name of the file you wish to have it saved as. On line 32 is where you have the option to load a previous test, otherwise leave this is an empty set of quotation marks “”.

All backups are saved at the end of every generation. These files are all saved in the same directory that the main lua code is saved in. To view these files go to the directory. Each file is saved in the following format: backup.[generation-number].[name-given].txt. An example of this format is backup.4.generationRandomMutate.txt and this is what would be entered on line 32 if you wished to start from this generation when running the program again. It will load the information on that population and start from this position.

All of these files can be opened and viewed in the text format as well but we recommend not making any actual modifications to the file as this could potentially cause errors when running or loading the file into the system.

You can keep these files as long as desired but often we recommend keeping only the most recent generations, the last five say, to keep as references or potential starting points. As the program runs, and depending on the size of your populations it could quickly create many files that would not all be necessary to keep. However all generations will be saved so it is up to the user in the end on which ones to keep and for how long.

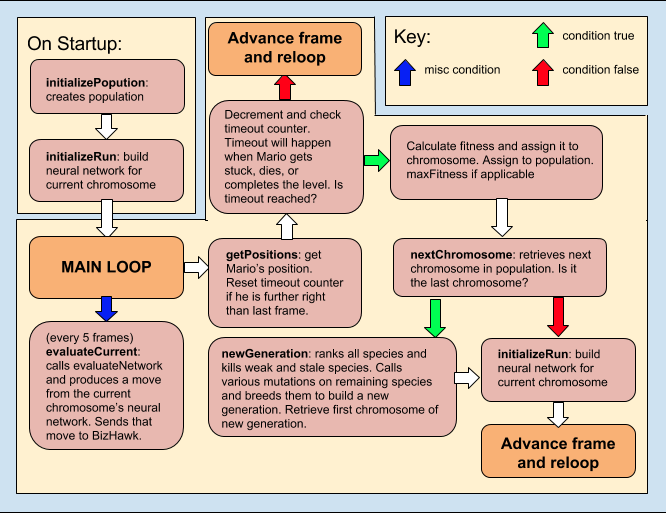
## Failure Recovery

If BizHawk is failing to open, or has become unsupported, then go to <http://tasvideos.org/BizHawk/ReleaseHistory.html> and download the newest prereqs file, and the most updated BizHawk file. Then repeat the Installation steps with the newly downloaded files.

# Programming Guide

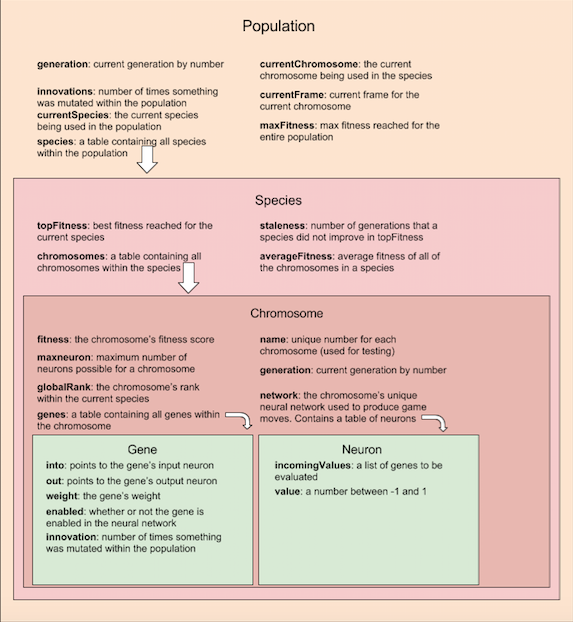
## Software Structure Chart

This chart illustrates the the main algorithm in the system. Smaller parts are abstracted out from this chart.



## Data Descriptions

This system does not use a database. All of the data used by the system is stored locally during the system’s runtime. All data is contained within a population table variable.



## Source Code Catalog

All source code for the system is contained within a single lua file. Chunks of the code are divided into sections based on their functionality. The lua file documents each section’s primary author, its purpose, and its last date of modification.

|  |  |  |  |
| --- | --- | --- | --- |
| **Purpose** | **Line Number** | **Last Modified** | **Author** |
| Data creation. Creates the initial population and all data inside of it. | 60 | April 6 | Christopher Bruns |
| Gene creation and copying functions. | 144 | March 7 | Christopher Bruns |
| Creation of the neural network and neural network utility functions. | 174 | April 6 | Jack Van Gent |
| Crossover mutations. Contains crossoverMutation, onePointCrossover, and twoPointCrossover. | 275 | April 15 | Christopher Bruns |
| Node mutations. Contains nodeMutation, tailNodeMutation, twoNodeMutation, and stepMutation. | 445 | April 22 | Christopher Bruns |
| Link mutations. Contains linkMutation, doubleLinkMutation, and mutation utility functions. | 625 | April 20 | Christopher Bruns |
| Base mutations. Contains functions to mutate chromosomes in various ways. | 805 | April 22 | Christopher Bruns |
| Interaction with BizHawk. Contains functions to retrieve the state of the game from the BizHawk emulator. | 1041 | March 22 | Jacob Espenscheid |
| breedChild function. Breeds a child chromosome using the crossover mutation. | 1145 | April 13 | Christopher Bruns |
| altBreedChild function. Breed a child chromosome using a random crossover mutation. | 1169 | April 25 | Christopher Bruns |
| Neural network evaluation. Functions that evaluate a chromosome’s neural network to produce a button press. | 1205 | March 29 | Jack Van Gent |
| Functions that happen between generations. Functions that clean up the current population, call various mutations on the population, and create a new population for the next generation. | 1282 | March 22 | Jacob Espenscheid |
| File IO for data backups. Saves and loads files containing information on the population. | 1483 | April 16 | Jacob Espenscheid |
| File IO for test data. Outputs data regarding the current run to a text file for examination of the current settings. | 1580 | April 2 | Jack Van Gent |
| Data initialization. Functions that call the data creation functions at the beginning of the program. | 1657 | March 7 | Jacob Espenscheid |
| Main loop. | 1788 | April 24 | Jack Van Gent |

## Test Data

Our main goal of testing was to figure out which mutations and settings yielded the fastest way for our system to learn and complete levels. To do this we fashioned a system that would output data to a text file regarding the system’s progress. When the system reached a new fitness level, a timestamp would be outputted to the text file. We would run the system multiple times with different settings and mutations, and compare the timestamps between runs to see which mutations and settings performed best at different points in the level. Our test files are all documented and saved in the directory “Test Files”. At the top of each test file will be a short description of what that file was testing.

We found through our testing that having a variety of options usually resulted in the better performance and time of completing the level. While all of the alternate variations when tested individually with the original mutation types were able to perform and eventually complete the level some did seem to outperform the others. The tailNodeMutation specifically seemed to perform well when used with the other original mutations.

However the other mutate functions appeared to work best. By allowing the system to randomly choose between all of the crossover, node, and link mutations when playing levels it added a lot more variety and the speed it took to complete a level decreased. This is also the same for when we applied the generation function to determine the types of mutations that were performed during which speed of completion was faster than when only a single type of mutation was changed.

By allowing our population to have more variance and for a larger variety of mutations our performance improved. The examples of some of our test data files that were created help to show how these milestones were reached at specific points when tests were ran.