[[1]](#footnote-1)

Hybrid Point-by-Point Wisdom of Crowds/Genetic Algorithm Approach with Super Mario Bros. NES

Hamilton, Jonathon T.

Computer Engineering and Computer Science Department

University of Louisville

December 6, 2016

*Abstract*— Super Mario Bros. NES has been shown to be an NP-complete problem. This paper proposes a hybrid approach blending the standard Genetic Algorithm with a Wisdom-of-Crowds (WoC) based algorithm, specifically implementing a character-by-character WoC solution into each population of solutions, and determines the efficacy of such an approach. The results of this technique are described herein. It was found that a character-by-character WoC method did not increase the efficiency of the evolutionary algorithm due to a loss of productive sequence information.

# INTRODUCTION

T

HE Genetic Algorithm is a method often used for solving NP-complete problems in a simple and straightforward fashion. However, it is not often the most efficient algorithm for solving these problems. But, because it is easy to implement, it can be easily modified and experimented with.

.

Wisdom of Crowds (WoC) is traditionally considered an effective method of approximating solutions to many problems where the algorithm is applicable. The WoC algorithm recognizes that, even if a group of people is not particularly expert at solving a problem, the collective average of their solutions is often more effective than any one individual (Surowiecki, 2004). The ability of WoC to be an effective algorithm depends upon the problem’s cognitive diversity (each individual has only its own information), independent origination of individual solutions, decentralization of knowledge, and ability to be aggregated (Surowiecki, 2004).

The problem I introduce is the video game *Super Mario Bros* for the Nintendo Entertainment System. *Super Mario Bros.* is a side-scrolling adventure game released in 1985 by Nintendo Co., Ltd. In it, the player, acting as the character *Mar*io*,* moves through a side-scrolling world, either avoiding or jumping on enemies, jumping across pits, and collecting coins often hidden within blocks floating in the air along his path.

It is well-understood that classic Nintendo games including *Super Mario Bros* are NP-hard (Aloupis, 2012). The purpose of this experiment is to determine if an efficient solution can be found using a hybrid GA/WoC solution to World 1-1 in *Super Mario Bros NES*.

This experiment runs solely with the Lua programming language, version 5.2.4. It utilizes the FCEUX emulator to run *Super Mario NES*, chosen due to its easy-to-use Lua API. The GA and WoC algorithm, mutation and crossover algorithms are all written in a single script. My approach is described in Section 3.

# Prior Research

Yampolskiy and El-Barkouky showed that a hybrid GA/WoC approach can increase efficiency in solving the Traveling Salesman Problem (TSP) by 6-9%. (Yampolskiy and El-Barkouky, 2011) The GA used was a typical evolutionary algorithm with a simple swap mutation and two different crossover methods, applied to TSP. Of particular interest here is the WoC aggregation method used. The aggregation method of a WoC algorithm determines the process by which the crowd’s solutions are aggregated together into a single “wise” solution.

Yampolskiy and El-Barkouky used an aggregation method that retains local connections between nodes in TSP if they are shared by 90% of individuals in a generation, and which always replaces intersections between pairs of nodes with parallel connections.

Togelius, et al. ran several GAs on *Super Mario Bros NES* and compared the results, including one solution using *HyperGP*, a neuroevolution/genetic programming hybrid method. (Togelius, et al, 2009). However, none of these methods include WoC as a possibility for a combined approach, or as a stand-alone algorithm for solving *Super Mario Bros NES*.

# Proposed Approach

The bulk of the script is set up as a genetic algorithm script. The basic GA used is influenced by Jason Brownlee’s Genetic Algorithm for bit string optimization, written in Lua, but modeled to *Super Mario Bros NES*.

In my experiment, each population held 15 individuals. Individual solutions in my project were represented as 500-length strings, composed of the numbers 1-4 that each represented one of 4 actions: moving right, moving left, jumping, and “runjumping” to the right (wherein Mario both moves to the right and jumps). The mapping of each character in the string is shown below.

Table 1: Possible inputs for GA/WoC *Super Mario Bros. NES*

|  |  |
| --- | --- |
| 1 | Run right |
| 2 | Jump |
| 3 | Run-jump right |
| 4 | Run left |

The mutation function used simply checks every character in the input string one-by-one and, if mutation occurs, exchanges the number at the current place in the string with another number, with an equal chance of becoming any of the other numbers. The mutation rate used in this experiment was 3%; meaning that 3% of the characters in the input string would be mutated and replaced with another character. This works out to 15/500 characters in the input string changing per generation after crossover is applied.

The crossover function used is a simple double-point crossover. Uniform crossover was considered for this experiment, but ultimately was found to not be as effective as double-point due to the nature of each solution being represented as an input string. Double-point crossovers allow for identical segments of events to happen in-game from parent to child, maintaining some of the fitness of the parent depending on where the partitioning of the string occurs. A uniform crossover applied to a problem such as this would introduce so much random variation that it would be essentially useless.

The fitness function I used took two main things into consideration; the player’s actual score and the distance traveled throughout the level. Distance traveled was calculated by adding the value of a counter in the game file’s RAM that tracked Mario’s x-position on the screen, ranging from 0-255. When this value reaches above 250, a global distance variable is incremented. The reason behind incrementing this variable at above 250 rather than at 255 exactly is because the x-position counter often skips 1-4 numbers when Mario is moving. However, because this skipping occurs consistently throughout each and every run, there is no loss of accuracy in the final score.

After the current individual “dies”, either by running into an enemy or falling into a pit, the distance is calculated by multiplying the global distance variable by 255 and adding it to the current value of the x-position counter in RAM, which represents the remaining distance the player has traveled since the last time the global distance variable was incremented.

Additionally, after the current individual’s death, the player’s actual score in the game was added to the calculated distance, and this became the fitness of that individual solution. The complete equation used to calculate fitness in my source code is as such, with *score* being previously defined as the game’s actual score at the point of death:

d*istance = (globalX \* 255) + memory.readbyte(0x071C)*

*score = score + distance*

The Wisdom of Crowds algorithm used is as such: the program iterates through each input string in a given generation and determines which number is used most often at the current place being examined. Whichever number is used most often among all strings in the generation will be placed into a new string. After the last character in every string has been examined, the new string has been fully completed, and the string is inserted into the new population. This process is demoed below.

Table 2: Example of WoC aggregation method

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| String: | A | B | C | D (aggregate) |
| 1st character | 1 | 2 | 1 | 1 |
| 2nd character | 4 | 3 | 3 | 3 |
| 3rd character | 3 | 2 | 1 | 1 |

If two or more characters occur at a certain place in the strings with equal frequency, sequential order is prioritized. That is, 1 is prioritized over 2, 2 over 3, and 3 over 4.

# Experimental Results

Due to the lack of a common standard fitness function among *Super Mario*-based evolutionary algorithms, as well as the fact that previous *Super Mario* evolutionary algorithms used different versions of *Super Mario* (Togelius, et al using *Infinite Mario*, based on *Super Mario Bros. 3*, and SethBling’s NEAT Mario[[2]](#endnote-1) used *Super Mario World*), it is difficult to objectively compare the results of one study versus another.

## Data

The data used in this experiment are simply the environment of World 1-1 in *Super Mario Bros NES*, including enemies, blocks, pits, and other obstacles. Because *Super Mario Bros NES* is a side-scrolling game with an absolutely constant environment, even including enemies, the data used in this experiment are exactly the same for each run.

This is ensured by the use of *save states*, which are tools often used in video games used on computer emulators. A save state is a location in-game that can be saved, either with the use of an emulator control or with the use of the *savestate* class in FCEUX’s Lua API in our case. This location isn’t just the player’s physical location in the game, but also includes all information about the game at the point in which the save state was created; the player’s score, number of lives, location of enemies and obstacles, current time remaining, etc. It is an exact copy of the state the game was in when the state was created.

This savestate function allows each and every individual in our program to run with the same environment, because every time the player dies, the same savestate is reloaded and another individual begins running.

## Results

The improvement in fitness over 51 generations running a hybrid GA/WoC approach on *Super Mario Bros NES* is shown below.

Figure 1 Improvement Curve of GA/WoC approach on Super Mario Bros. NES

While this data shows that an improvement has been made over 51 generations, it doesn’t show whether this is due to WoC, or simply the GA working as intended. Shown below is a sample of 5 generations comparing the aggregate input string with the fittest input string of each generation (at no point was the aggregate WoC string ever the fittest string of its generation):

Figure 2 Comparison of Aggregate WoC individuals with GA-driven fittest individuals from each generation

This demonstrates that the WoC aggregation method I used was not nearly as efficient as a standard genetic algorithm.

# Conclusions

While Wisdom of Crowds has great potential for improving NP-hard problems other than *Super Mario Bros NES*, the data shows that WoC did not increase the efficiency of a standard genetic algorithm.

Why is this? It appears to be a result of the aggregation function moving about character-by-character, as opposed to retaining successful sequences of inputs. The character-by-character aggregation method retains individual inputs from successful solutions, but fit individuals are created by sequences of inputs at ideal times, rather than the inputs themselves. As a result, the character-by-character aggregation method used in this experiment acted as little more than a roundabout way for increased genetic variation to be implanted into each generation.

# References

1. Surowiecki, J. (2004) The Wisdom of Crowds: Why the Many Are Smarter Than the Few and How Collective Wisdom Shapes Business, Economies, Societies and Nations, Little, Brown.
2. Yampolskiy, Roman V., and Ahmed El-Barkouky. "Wisdom of artificial crowds algorithm for solving NP-hard problems." *International Journal of Bio-inspired computation* 3.6 (2011): 358-369.
3. Aloupis, Greg, et al. "Classic nintendo games are (np-) hard." *CoRR, abs* 1203 (2012).
4. Togelius, Julian, et al. "Super mario evolution." *2009 IEEE Symposium on Computational Intelligence and Games*. IEEE, 2009.
5. Brownlee, Jason. 2009. “A simple genetic algorithm for function optimization, in lua.” GitHub repository, <https://gist.github.com/jbrownlee/247952>
6. Genetic Algorithms – Crossover. Tutorials Point. [Online]. Available: <https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_crossover.htm>. Accessed November 28, 2016.

1. [↑](#footnote-ref-1)
2. MarI/O – Machine Learning for Video Games – online at https://www.youtube.com/watch?v=qv6UVOQ0F44 [↑](#endnote-ref-1)