An attempt at optimizing solutions to Super Mario Bros

Filipe Freire
FCT, University of Algarve, Portugal
a85493@ualg.pt

ABSTRACT

This report revisits Leane's 2017 evolutionary strategy approach to optimizing solutions for classic Nintendo Entertainment System (NES) video games [1], focusing specifically on the first level of Super Mario Bros (SMB). Recognizing NES games as computational problems, we attempt to reproduce and adapt Leane's methodologies with an instructional focus. The work involves interaction with a game emulator, developing a representation for potential solutions, and employing metaheuristic algorithms for optimization. Our study, while less comprehensive than typical research, provides insights into the basic workflow of these algorithms as well as results we achieved in our attempt to reproduce the original paper. Note: All code developed in the context of this report is freely available in an opensource repository on GitHub [2].

KEYWORDS

Metaheuristics, Genetic Algorithms, Videogames, Super Mario Bros

> *"It's a dangerous business, Frodo, going out your door. You step onto the road, and if you don't keep your feet, there's no knowing where you might be swept off to." – J.R.R. Tolkien, The Lord of the Rings*

1 INTRODUCTION

The primary goal is to develop a solution for the first level of Super Mario Bros and optimize it using evolutionary strategies. We divide the problem into manageable tasks: interacting with the game emulator, representing potential solutions, generating initial feasible solutions, and optimizing them. Our work navigates through various roadblocks and nuances in applying evolutionary strategies to video games.

2 THE PROBLEM

Reading through the Leanes's 2017 article [1] we find that it appears to be possible to use old Nintendo Entertainment System (NES) [3] videogames as "sandboxes" to experiment certain metaheuristics algorithms.

That article [1] presents a evolutionary strategy approach [4] of what it would take to optimize solutions to a few of these retro videogames. For different types of videogames within the NES the authors walk us through a flow that consists in generating a feasible initial solution and then iteratively attempting to improve that solution with a custom population-based approach.

There's a handful of literature on Nintendo videogames being NP-complete (see [5] and [6]), with the gist of it being: to achieve a "victory" state in those videogames, or even just the completion of a round or a level, we can input a sequence of input combinations on the gamepad controllers that will be a solution for the game or levels within the game.

So in this work, we make an attempt at reproducing the original paper [1], with a smaller instructional scope.

The proposed goal is, like mentioned in the introduction, of at the very least, being able to get to a solution for the first Super Mario Bros (SMB) level, and to then try to optimise that solution. The primary objective of all this being to understand the basic workflow of the subject matter presented in [1], rather than delving into the intricate depths that would be typical in a similar but comprehensive research scenario.

3 SPLITTING THE PROBLEM

Reading through the Leanes's 2017 article [1] it became clear that to come close to reproduce it we need to split the whole endeavour into smaller tasks. There will also be other tasks not explicitly mentioned in the report. The main ones we came across were the following:

- Interacting with the videogame emulator: being able to read values from the game and perform inputs on it.
- How to represent potential solutions: what do we store and how?
- Getting an initial feasible solution: following the article's approach and using a cyclic way of going through motifs to get a solution that can complete a level in Mario.
- Optimizing the solution: performing mutations, crossover and repair on solution segments that don't perform well.

3.1 Interacting with the game

First off, and not necessarily expressed in the original article, we'll need to find a way to interact with the game.

For running the game we'll use an emulator. The same kind of emulator that is referenced in [1], FCEUX [7].

There are more NES emulators, but this one in particular is appropriate for our work for two key features. The first is it has a Lua (the programming language) scripting layer [8]. That enables the following:

- We can mimic any specific combination of gamepad/joystick inputs at each drawn frame of the game.
- We can read the position of where the Mario character is from a RAM memory address of the game, provided we know the memory addresses to look into [9] (See figure 1)
- We can also read if the player character has died or fallen into a pit, or infer from the position if they are stuck at a part of the level or have hit some sort of local maximum.
- We can define save states and easily restart back to a save state and restart again a given path attempt we are following.

The second key feature of FCEUX is that we can increase the emulation speed to play the game at more than its usual speed, so we can test out different solution attempts faster. This is both a blessing and a curse. Unlike with other metaheuristics example

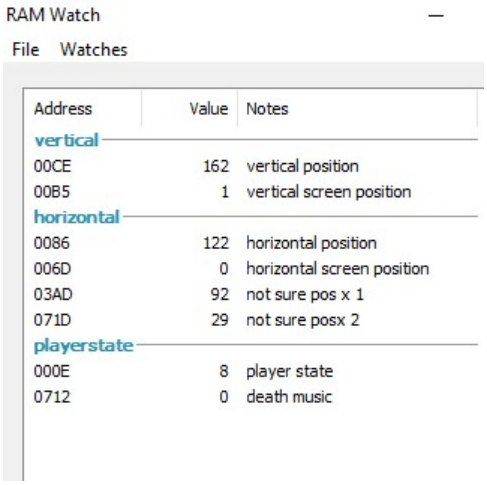

Figure 1: Example of watching values of some RAM addresses in FCEUX for SMB

problems where we can maybe resource to parallelism for mutating a large population of solutions and checking their fitness in code, in this case we are conditioned by how long the emulator will take to run through the solution to check its fitness.

3.2 How to represent a solution

We can split the solution space into different *motifs* . These *motifs*, read, *patterns*, will be NES Gamepad input combinations. A typical NES Gamepad [3] has 8 buttons the player can press (or not) simultaneously. So our solution can pick from can have a maximum of 2^8 (roughly 256 possible motifs). We don't necessarily need all of these motifs. Only a portion are useful in getting our player character across a level, so we'll initially narrow the motif possibilities down.

For representing the motifs in a String *motifKeys* we defined the following motif keys:

```
{"right", "rightA", "rightB", "rightAB", "left",
"leftA", "leftB", "leftAB"}
```

Each motif key would represent a Gamepad input combination, like the example bellow where we only simulate press the right Gamepad button as well as the Gamepad's *A* button:

```
motifs["rightA"] = {
    up = false,
    down = false,
    left = false,
    right = true,
    A = true,
    B = false
}
```

There's a difference here from the original article, in order to help readability of a given solution, the motif keys are written expressing the Gamepad input combination, as opposed to just code-like motifs like <1, <2, <3, ...

Attached to the motif we'll also need a value to represent how many frames we are attempting that combination:

```
local frameDurations = {10, 20, 30}
```

With these elements we can represent solutions strings. We've done so like this:

```
rightA:20, rightB:10, right:30, leftA:10, ...
```

3.3 Initialization

The second problem has to do with generating an initial feasible solution. The article talks about 2 methods: cyclic generation, another random generation.

For brevity, for now we'll focus on the first one. The general idea of the proposed algorithm is the following:

- 1) We fill an initial solution attempt with motifs of the same value, e.g. `A86 C` : 10 for a length n.
- 2) We try to run that attempt and see how far we got and if the player character dies or got stuck at any motif.
- 3a) If our player character got stuck or dies we try to increase the duration of the motif and/or switch to next motif on the list of possible motifs.
- 3b) If we were able to run through all the motifs without getting stuck or dying but we haven't reached the end of the level we append another motif to the solution, increasing it's length n by 1 and try again.
- 4) We continue doing this until we reach the end of the level.

Once Mario reaches the end of the level we save that solution as a feasible solution to a file. This can then be read and used as an input solution into the population-based search algorithm which will attempt to perform mutations to it and try to optimize it.

3.4 Optimization

The original article [1] basically shares the pseudo-code, in broad strokes, for a custom population-based algorithm approach in order to handle optimization of solutions. The understanding we achieved of the underlying custom logic for the optimization step is the following, in general terms:

- 1) Take a feasible solution $($.
- 2) Find segments where heuristic score is low.
- 3) Mutate the motifs in those segments with random motifs.
- 4) Crossover different segments into a new solution $($'.
- 4.1) Check if solution is feasible (reaches end of level) - if not, repair it.
- 5) If $($' is "fitter", use it in 1) and repeat until termination criteria.

A few things might make the reader's curiosity senses *tingle*: What does it actually mean when we say a solution is feasible... does it mean Mario reaches the end of the level? Or perhaps even if Mario does not reach the end of the level, as long as there is some sort of improved fitness value, that makes it feasible? What does it mean to repair a solution? Do we repair it to a point where the solution completes a level or merely has an improved fitness value?

It's tricky to derive, through intuition, what's the answer to most these just from reading the original paper. And as will be mentioned in a later section, these questions ended up being actual traps / choke points.

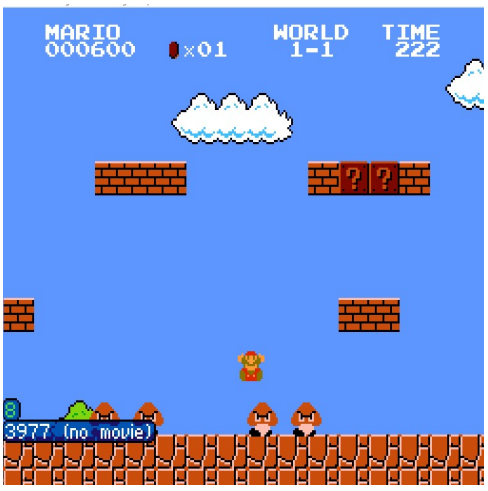

Figure 2: Example of frequent failure spot where Mario can get easily stuck dying between two groups of enemies close to each other.

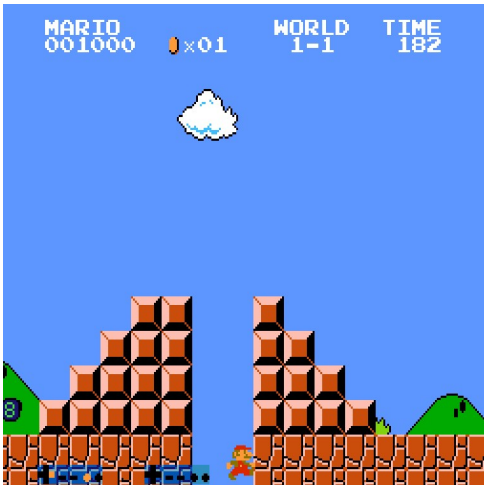

Figure 3: Example of a frequent failure spot where Mario can easily fall into the pit

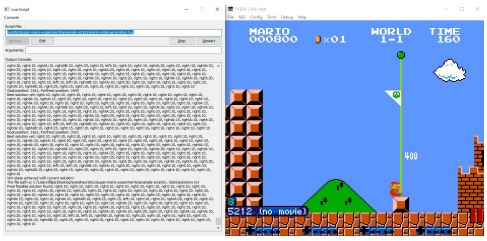

Figure 4: Example of reaching a successful feasible solution where Mario reaches the end of the level.

4 RESULTS / ANALYSIS

4.1 Roadblocks faced while getting an initial working solution

Starting with the algorithm we use for initialization. Some changes had to be done in the actual code that makes the algorithm slightly different than the one proposed in the article. This is due to a handful of problems:

- Getting stuck between enemies that are too close together (for example, see Figure 2).
- Getting stuck aggaist a "wall" where maybe we need to go back a bit before jumping forward.
- Having to jump for a different time duration to hop over different wall sizes.
- Having to jump in small hops (or with some acceleration) to avoid falling into some pits/holes. (for example, see Figure 3)

One workaround added for some of these problems were to try to change the motif previous the actual motif where the failure happens. This helps in cases where Mario needs to already be executing a helpful motif before the spot where we run a failing motif (e.g. like be in a jump state slightly before a pit/cliff's beginning).

The other workaround had to do with cycling through different frame durations of a given motif. That would allow that gamepad input to be pressed for a longer time and would accommodate for Mario doing taller or longer jumps for example.

Generating a good solution, even with these workaround, can take a long time and many iterations. This might explain why the original author would then try to pick a random motif when trying to fix a failing motif as a strategy opposite to cycling sequentially through motifs.

After workarounds we managed to reach an initial feasible solution (see Figure 4). The record time for completing SMB first level is roughly **29 seconds**. Our un-optimised initial solution took about **42.5 seconds**. *Not too shabby.*

4.2 Massive roadblocks on Optimization, *hic sunt dracones...*

Here be dragons. After attempting to implement an optimization algorithm following the steps laid out by the original article [1], there are a handful of *dragons*:

- Lack of meaningful detail into the heuristic function used to pick segments to mutate
- Weirdness/confusion related to Fitness, e.g: euclidean distance vs. completing level
- Article talks about using a population of 3, but we found that is not nearly enough to accommodate for this "evolutionary"-like algorithm that they propose. This will be discussed in the next subsection.
- And a handful more issues...

Even not considering the population bounds weirdness, there's feasibility weirdness. When we attempt to follow the suggested optimization algorithm to the letter it doesn't provide interesting results when it comes to better solutions. This is both in terms of how much time it takes to get to a useful or meaningful mutated solution as well the very few solutions that are feasible (in our understanding, that complete the level) are not necessarily "faster" than the initial feasible solution we throw at the algorithm.

Attempting to salvage that algorithm by skipping the suggested crossover and repair stages also doesn't seem to help. It helps a bit in the amount of level-completing mutated solutions produced but we still face the same hardships as the previous point, in that the vast majority of mutated segments tend to help Mario getting stuck or failing to reach the end of the level, as opposed to actually helping Mario finish the level sooner. This will be discussed further in a follow-up subsection.

The major consequence of these issues is that we ended up not being able to produce enough meaningful data using this algorithm that would allow us use to sample and perform statistical analysis and safely draw some considerations regarding its performance in this context.

It might be the case the implementation we arrived at is not nearly in the same working condition as the one in the original article, and since we don't have access to the code, we can guess what might be causing most of these issues. We'll share some more in-deep observations in the next subsections.

4.3 Population sizing problems

There's a major problem and pain point with the original article: the size of the population. The article mentions a population of 3, and narrows it down to the fact that at the time it was computationally expensive to do a bigger population.

What we found in our implementation attempt was that that particular size of population was not nearly enough to even get past one iteration where we do a whole flow of of the custom "mutation, crossover, repair and selection" algorithm.

In some cases it was necessary to cycle through hundreds of mutation alternatives, even before crossover, just to make sure that the proposed mutation was feasible in reaching the end of the level.

It felt at times that the article had a bit of an accidental oversight: we mutate a segment, but we don't consider when mutating the fact that the mutation might make the solution infeasible at reaching the end of the level.

Instead the article suggests we go ahead with a mutation and crossover of motifs of mutated solutions and then have an *Repair* step after crossover. In practice this was proving worse during execution trials than the cyclic generation (which is itself kind of a blind search?) for the feasible initial solution. We would end up stuck for a long time trying to repair a solution that was a crossover of multiple mutations for a longer time than it took to get to an initial feasible solution with resource to cyclic generation at the beginning of the problem.

One thing that was attempted was to try and not enforce any bound of the population size [10], and just cycling through as many working solutions as possible, to try to understand a reasonable population bound. We didn't arrive at a meaningful conclusion in this particular last point.

4.4 Trying to salvage the optimization

We learned that in some cases the emphasis in evolutionary strategies is more on mutation than on crossover [4]. So we tried performing mutations of segments of motifs, skipping the crossover and repair. Still no success.

As a light side-note: the crossover suggested by the article didn't "spark joy". Plainly put, it was something in the lines of... *just take all mutated segments and place them in a solution, if parts of a segment that was mutated are shared by some other segment, keep the first one...* yeah, didn't really *spark a lot joy*.

The suspicion at this point was that: there was likely something wrong with our own implementation of the checking the heuristic value of segments. The gist of it being the heuristic value being calculated in a form that doesn't help in picking good segments to mutate, regardless of the parameter values used for the threshold to pick lower heuristic value segments. It's hard to know for sure what is wrong since the original article only provides some clues into how to implement each part of the optimization algorithm.

The whole mutation process where even in mutated solutions that reach the end of the level, the amount of motifs of those solutions remains the same as the initial feasible solution, when we would expect that some motifs could be redundant after mutation.

4.5 *Going for broke* and *rolling the dice*

From all the problems mentioned in analysis we have to try to change course if we want to at very least perform some sort of statistical study and performance evaluation on a metaheuristic and salvage some part of this whole effort under the time constraints at hand for delivering the report.

One potential candidate is to turn our focus into the task of achieving a first feasible solution.

The initial solution we got at this point was following the cyclic generation algorithm that is presented in our article under study [1]. From observing a couple of attempts of running this algorithm we ran into a couple of hick-ups.

Like mentioned in the article, the cyclic approach doesn't always help in getting our player character unstuck from a place or creatively avoid some enemies.

The article points out that we can alter that cyclic generation and replace motifs where we are failing with random motifs (and vary the duration randomly) to try and progress the player further through the level. This would make our algorithm closer to a uninformed random walk, which would at least enable us to evaluate it and compare different runs with different random seeds.

We went ahead and implemented the following changes:

- 1) We fill an initial solution attempt with motifs of the same value, e.g. `A86 C : 10`.
- 2) We try to run that attempt and see how far we got and if the player character dies or got stuck at any motif.
- 3a) If our player character got stuck or dies we change the failing motif (or the motif previous to the failing one in some cases) with a **random motif and random duration**.
- 3b) If we were able to run through all the motifs without getting stuck or dying but we haven't reached the end of the level we append another random motif to the solution and try again.
- 4) We continue doing this until we reach the end of the level.

With this kind of approach, and controlling only the random seed as an imput parameter, we achieved results expressed in Figure 5 by doing 30 runs with different random seeds. We can also see the standard deviation in Figure 6.