John Pugmire, Aleksa Perucic, Peter DeBrine, Liam Yoo

Ode to a Hill Climber

AI Final Project
C-term 2017

Abstract

With modern Artificial Intelligence approaches focusing heavily on learning over a long period of training, we turned to a decades-old problem space with a decades-old approach: hand-parameterized hill climbing. Lua scripts were written to control Super Mario Bros in an NES emulator called fceux. Because of the nature of the scripting engine and the limited development time, this made it very difficult to pass information in and out of the script while it was running, so we were restricted to algorithms with short learning periods and no need for training input. Several scripts of different approaches were written to control the game with the simple goal of finding any path to the solution by any means necessary. Some hill climbing approaches were found to be rather effective at finding solutions to almost all levels of the game, while more sophisticated algorithms involving annealing and genetic breeding were found to fall short, though this may be due to a bad representation of the moves which Mario can make.

1 Introduction

Our project was inspired by a fondness for Mario coupled with influence from other AI's seen on the internet. Inspired by the blind amnesiac metaphor, we represented this by our algorithms not requiring an extensive training period nor any input data. Another factor driving this change was due to the time constraints, as opposed to implementing a type of vision, Mario looked at specific memory addresses, there was a focus on minimizing the amount of lookups, as a human player would not have access to that data. Key aspects of the game, such as X position of Mario, as any human player would understand. While this lead to a lot of time being devoted to watching Mario run into the same Koopa Troopa perhaps over ten thousand times, it became extremely satisfying to see Mario bypass that previously impossible task over and over again.

2 Successful Approaches

Every successful approach had several things in common. First, they all use a time slice of 5 frames in the emulator as the length of one 'move', which is simply a combination of keys. In each approach, the keys for a move are generated from a table of probabilities which we hard-coded. The function for generating a move considers which direction to travel in, and

whether to press the jump button. There are 4 parameters: pRight and pLeft, which tell the respective probabilities of the player walking right or left, pJmp, which is the probability of pressing the jump button if it's not already held down, and pJmpHold, which is the probability of continuing to hold the jump button if it's already pressed. As a result, the length of each jump follows a Poisson distribution, which proved to be useful. In the code, there's also a parameter called pRun, but this value is not used.

Search with backtracking and block snapshots: This solution was inspired by a few observations in the game. First, that after you make it a certain distance once, you can always make it that far again. Second, since we don't care about how quickly we win a level, there's no reason to test multiple paths to the same point in the level. Third, in Super Mario Bros, the screen doesn't scroll backwards, so if you can make it one entire screen length past a given point, then that means there's guaranteed to be a path to the end of the level from that point using the sequence of moves with which we reached it.

We took advantage of this to implement a feature called "blocks". This meant specifying a block size-- in our case, just over the length of the screen-- and adding logic to never backtrack further than the block length. In addition, we made use of the emulator's save state features, greatly reducing the amount of time that each trial took. All together, this created a powerful AI that can play through much of the game.

Search with iterative backtracking: Because the exact location of the snapshots was more or less random within a certain area, we occasionally got stuck because a less-than-optimal path was used to reach the block snapshot location, and it would take many trials to make any progress beyond that point. The iterative backtracking approach was meant to address this problem, by starting out with smaller amount of backtracking and increasing it as there were more failures. Our results show that on most levels, this solution was more performant than the non-iterative approach. However, it did not really have the intended effect of making it easier to deal with bad block locations. In fact, it couldn't finish level 1-3 nearly as consistently as the previous algorithm, which indicates that it was a failure at its intended goal.

Search with iterative backtracking and "entropy": This approach was an attempt to incorporate some intelligent behavior into the algorithm, while still not relying on level-specific information or hardcoding strategies. The reasoning is, if we fail to progress for several iterations, in addition to increasing backtracking, we should also try something different.

The entropy value simply determined the likelihood of "mutating" the original move selection parameters before making each move. This simple mutation function was taken from a failed genetic algorithm. It functions by selecting a new value for either pJmp or for pRight and pLeft. Then, it updates those probabilities by performing a weighted average with the original value.

This approach was extremely successful and plays through the game with minimal difficulty until world 4, where it sometimes takes very long on level 4-2 and cannot possibly complete level

4-3, because it always gets stuck in a spot where you need to run and jump, and our script doesn't run. It also outperforms all the other approaches on level 1-3 substantially, which was the hardest level for which we collected extensive results.

3 Failed Approaches

Random search with backtracking on failure: The simplest approach we attempted was to simply generate a random sequence of moves until Mario died. At that point, we'd rewind a certain number of moves (it was chosen somewhat randomly) and try again. In order to avoid getting stuck, a dead-end detector was implemented. This approach solves the first level after a few minutes, and has been observed solving the second level, but overall is something of a failure.

Hard Coded Pathfinding: This algorithm attempted to solve the problem of Mario getting stuck on various parts of the level and enemies by reading the active screen memory and generating multiple paths that mario can traverse. This path generation would happen on every frame, and it turned out to be fast and efficient at finding a path for Mario to get through the level. However, the problematic part of this implementation was the translation of these path coordinates into Mario's movement on screen. Specifically, if we were to get Mario to move to a specific part of the map, due to the way physics in Mario is handled, he would frequently overshoot his x destination. This would occur since Mario's x velocity wouldn't immediately stop once the target block is reached, causing it to move to the next block, which would confuse the algorithm. The problem was also that movement patterns such as jumps would have to be hardcoded in, which is both tedious and inaccurate. With more time, this algorithm could be combined with another learning algorithm that could learn to map Mario's movements with the path, but there was not enough time for this implementation to materialize.

Genetic Algorithm: Several genetic algorithms were created based on the parameterized move selection in the successful attempts. These algorithms all attempted to find an optimal value for those parameters for a given level. However, it turned out that this was a very, very slow and inefficient process. Even with a population size as low as 100, it could take several minutes to run enough trials to determine fitness before updating the population. And then, the generated parameters were still rather ineffective. Notably, the genetic algorithms tended to generate very low values for pJmpHold, which made it so Mario only ever made short hops. This turned out to be very good for avoiding enemies, but made it impossible to progress over pits. This seems to suggest that a different representation for moves would be necessary for the genetic algorithm to be effective.

Simulated Annealing: A simulated annealing approach was attempted, using a representation of concentration of deaths in proximity to Mario. While it seemed to do a good job of quickly coming up with solutions to places it got stuck, it proved quite buggy and difficult to tune, and due to time constraints and the creation of other working intelligences, lead to the discontinuation of this algorithm.

4 Data and Results

Data were collected for the three successful algorithms on the first four levels of the game. In order to somewhat-accurately determine variances and standard deviations, 10 trials were performed for each level/ai combination. If a trial ran for more than 10 minutes with no solution, it was abandoned, though this only happened for the iterative backtracking algorithm on level 1-3.

The normal backtracking algorithm can be seen as a baseline, since the other two algorithms were based off of it. On levels 1-1, 1-2, and 1-4, the iterative backtracking approach performs somewhat better in terms of average computation time, though it fails miserably on level 1-3, which is quite different from the other levels. This suggests that the algorithm was somewhat too specialized to handle a wider range of levels.

The entropy algorithm also tended to require less computation time than the original algorithm, except for on level 1-1, where it performs about as well. It also performs best by far on level 1-3, which demonstrates that it was quite effective in its goal of finding ways around difficult situations. However, on the flip side, if we consider the efficiency of the paths generated by each algorithm, the entropy algorithm actually did the worst, in that its final solutions required the most moves on average to finish each level, including level 1-3.

Otherwise, the three algorithms were somewhat similar. For more in-depth results, it would be good to collect data on the levels in worlds 2 and 3 as well, though the algorithms other than entropy tend to get stuck for a long time on levels 2-1, 2-2, and 3-3.

5 Conclusions

Creating any sort of intelligent artificial intelligence is hard, and adding time constraints on top of the high workload of a WPI student during the final weeks of classes. While we didn't have time to develop a more elaborate learning algorithm, the time constraints actually forced us to think outside the box. This lead to a successful foundation for our algorithms based on search with backtracking. The algorithm performs so well, that it is comparable to, if not better than, most other Mario solving AI's found online. If there was an easy way to get screen data to lua and

teach the AI through that, we would have probably never thought of the solution we did, and would have ended up with a lesser algorithm. This goes to show that hard work is good, but hard work based on a good idea is even better.