SIMULATION OF COMPETITIVE EVOLUTION THROUGH NATURAL EVOLUTION STRATEGY

Student: Igor Samurović (igorsamurovic@hotmail.com), SL23-13, Softversko Inženjerstvo (Software Engineering), Mentors: Đorđe Obradović, PhD, Ivan Perić, Mint, Faculty: Fakultet Tehničkih Nauka (FTN), eng Faculty of Technical Sciences, Course: Osnove Računarske Inteligencije (ORI), eng Fundamentals of Computational Inteligence

0) The basic premise

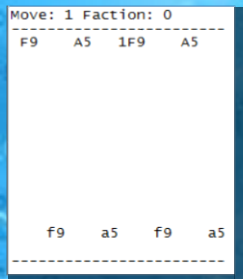
The goal of this project is to simulate real time competitive evolution. Depending on the number of options and adjustments to the main fitness function, it is possible to create a wide array of outcomes in this simulation game, despite retaining a similar heuristic

1) The Goal

Unlike what would be a logical goal of a similar project dealing with a finite state game like this, since we are evolving both factions during gameplay, it would be hard to implement a purely deterministic system of reasoning effectively. Therefore, the goal of this project is not to make an AI that will defeat any (even human) opponent, but rather, an AI that will defeat its evolutionary opponent and be able to keep with challenges continuously set by it. The project is implemented in Python 3.4 without any external modules.

Much like organisms evolve to exploit each other's weaknesses, so do the factions in this game.

2) The Game



The game that's specifically created for this project is effectively an oversimplified version of chess.

The game is played in a closed system, a 8x8 board in which four units belonging to each faction are placed in a symmetric top-down order. Each unit is at any time designated by three things:

1) its location on the board
2) Its type (A or a for Archer, F or f for Footman)
3) Its faction (letter size)
4) Number of hit points
5) Its attack damage and range (Footmen will hit every unit two fields from where they attack, in a particular direction, while Archers have the range of five. There is no friendly fire and the projectiles and attacks pass through own units (making it possible to use footmen as meat shields for archers, for example).
5) Whether they are performing an attack move in some direction (there are 8 directions, with 0 being bottom, 1 being bottom-right, etc). This number will be displayed next of the name when necessary.

3) The Approach

This section will briefly the approach in which the game mechanisms evolve generations and maximize the fitness function at every stage. There are two ways to create new data in this game, Mutation and Random generation. To begin playing the game from both sides, we'll use random generation to create moves since we do not yet have a reliable or specified model to follow. But moves have to be legitimate, meaning that we cannot expect a unit located at 0x0 to move in a direction that would require its coordinates to change to -1x-1, since that's out of bounds.

Instead, we'll use random generation on two occasions – when we first need to generate a move since we do not yet have sufficient data (the number of moves we've used until now exceeds the models' number of moves, or we have no model yet to begin with) or when we are encountered with trying to perform a move that's suddenly become illegal. This happens when the opposing faction evolves in such a way to block what would've been a legal move for the observed faction. Since it is not possible to forfeit a move, we'll randomly generate a new move in its place, but attempt to follow the model through afterwards. One mutation in the opposing faction can, by blocking the observed faction's moves, create a whole chain of events that ultimately lead to a huge number of newly generated moves, which is both a blessing and a curse – in the case of failure, that move might lead to a lost game, but can also trick the opponent in case the opponent wins too swiftly which makes it so that the fitness function prefers this game to others, even though most moves of the opposing faction might never be repeated again.

3.1) Mutations

Mutations are at a core of every evolutionary algorithm, and it is no different here. Depending on the setting of mutation probability, the mutation can occur in at least zero cases and at most half the cases. If the mutation setting is left at zero for one faction, it might win in the short run in case the opposing faction doesn't have enough time to mutate their own moves to fight it, but in 100% of tests that go well past 10000 games, the non-mutating enemy will definitely lose.

Aside from the mutation probability, mutation also occurs more in the faction that has a loss streak. It is calculated as:

MutationProbability * (1 + Faction.lossStreak)

This leads to more rampant mutations in the losing factor, as opposed to a more stable genome of the winning faction. The reasoning behind this is that, if the faction keeps losing, it needs to get as many chances to get away from the current genome, it is being forced to mutate a lot in order to win. And likewise, if the faction keeps winning, a high number of mutations can only deter it from winning consistently – instead, having a small number of mutations will make it possible for the winning faction to devise a better and more efficient strategy of winning, rather than cope with a loss. In this light, a bad mutation can make a winning faction lose the game or win it less convincingly, which will reflect in its fitness function and will likely result in a badly mutated, previously winning model, being discarded for good.

3.1) Model selection and fitness function

Each successful generation is saved along with its score generated by a fitness function, which will be explained shortly. A model is simply a set of moves that a newly created generation tries to follow, which it inherits from its most fitting predecessor. But the important part is that it's TRYING to to follow, rather than necessarily following. since an obstructed move can make all the difference in the game, as we've already covered before. At present state, after a lot of hard science (called trial and error, of course), fitness functions determines how good a generation is in this specific regard:

Fitness = (points * (1 + living/4)) / ((opponent.living + 1) + numMoves/ 100)

Naturally, points are an important factor here. We need to use the amount of inflicted damage to know how good our generation is with aggression, since defensive strategies are not the goal of this game.

The variable "living" refers to the number of living units that are left after the generation had been played through when it was saved. As said, this is not a defensive strategy game, but it is good to factor in the units left alive by the end of the game, because if we inflict the same amount of damage in two generations, it's better to prefer the ones where we have the more units standing alive, because in that case we are at less of a risk of losing more units than necessary in case the opposing faction develops a strategy against ours.

We are only factoring the opponent's living units in order to make it sensible to run games with restricted number of turns in order to force efficiency, rather than just the win condition alone.

And lastly, we are factoring in the number of moves in order to have the efficiency still have an edge over strategies that use more moves, since less moves lead to more stable (and less likely to mutate) generations, which is preferred if a faction is winning.

5) Testing

There are many testing parameters that can be fiddled with in order to produce a wide array of outcomes. There are two types of settings, game-specific and faction-specific.

5.1) Game-specific settings

It is possible to change the maximum number of moves per game. By doing this, we are forcing evolution to take place earlier, rather than later (it's possible for a game to go past 1500 moves). By setting this number to a relatively low number, we are forcing a lot of draw games in the beginning, so it may not be particularly useful in a short run. By setting this number to anything other than zero (which makes it so that there's no limit) we are skewing some games that might produce interesting results, so it is generally advised not to use any limitations bellow 1000 (games past 1000 moves are likely to be useless for the winning function since too much can go wrong, and fitness function is going to punish them severely to begin with).

It's also possible to change the number of games played. Up to around 1000 games, a faction that is not evolving at all (with MutationProbability set to zero) can still win as the opposing factor might not find a decent enough strategy to win in the 1000 games. At around 10000 games, the evolving faction will practically always win because there will be enough data (with a fairly decent MutationProbability like 0.01 down to 0.001).

5.2) Faction-specific setting

As far as factions go, it's possible to change two settings – MutationProbability and LookbackDistance. In practice, it appears that having anywhere between 1 and 10 changes per generation is going to be beneficial. But since MutationProbability does not limit the number of mutations that will occur in a generation, but rather, it is calculated for each specific move, it will change shorter move lists less frequently. If a winning generation has about 30 moves, MutationProbability of 0.01 will be extremely low. But even so, if there happens to be a winning move that has around 1000 moves, and is choosen by the fitness function because other aspects of it outweigh the enormous number of moves, MutationProbability of 0.01 causes 10 changes, and if they are particularly early, that can cause a chain of events that make it impossible for the generation to follow its model, often dooming it. Therefore, the more consistency a faction has in winning, the more likely it is that a lower MutationProbability value is beneficial, so for a faction to win after a certain number of games, it's necessary to have a large enough value to cause enough divergence in order to be able to develop strategy against the opponent, but at the same time, it needs to be low enough that once the faction is winning consistently, there are not many mutations that make it impossible to follow the model well enough. Some basic testing is done, which is why I have some anecdotal data that can be confirmed by running tests again, but a search for the "perfect value" would take a lot of resources since it'd require game sets with millions of generations to establish anything for sure.

Option LookbackDistance refers to the size of the list considered when looking for a model. Since we are always picking the most fitting specimen, and fitting values are already calculated based on the described formula, it restricts very old winning generations from being picked, despite their score, because if we don't cut off data that had become irrelevant, our faction will constantly keep trying an outdated strategy that does not fare well against the current opponent.

5.3) Testing output

Every time the testing commences, outFile.txt will be filling with information regarding game number, who the winner is, what the score is like, how many units are left standing, and what the worth (fitness) of a generation is. Likewise, the first and last as well as the game in the middle, will all be output to files called game1.txt, gameY.txt and gameX.txt where X and Y are the number of games played, and half that number, respectively.

6) Conclusion

At the most basic level, it is possible to simulate reactive co-evolution through generations, utilizing the randomness that spawns from mutations to create new genomes, and then letting a computational version of natural selection, a fitness function, perform keen selection among the generations.

While this project started off as an attempt to utilize Genetic Algorithm, it was soon apparent that an approach using it would only work well by indexing lists of favored specimen by the the data on board and trying many move sets against the same, not-yet-evolved enemy, and then not evolving while the opponent is attempting their own sets of moves. This approach is legitimate, but it would take way too much processing and data to be effective, so I had reduced the purpose of the project from making an AI that could eventually play this pseudo game against a human down to two warring factions constantly trying to evolve traits that let them defeat each other.

From an evolutionary perspective, natural evolution strategy is a much more realistic portrayal of how natural evolution works since the specimen may face a different opponent every time, rather than being given the luxury of attempting to combat the opponent with the same exact genes as many times as needed in order to reach an optimum. Because, even were that the case, the opponent also evolves, which makes the previous, highly specific solution that requires a huge amount of processing power – somewhat futile considering that the opponent will also be able to evolve in a way that completely counters that strategy. In this sense, Genetic Algorithm could as well be replaced by an A Star search method that would result in a back and forth match where the sides are equal, and as we can see in nature, a winner/loser equilibrium is quite a rare sight.

The selection system is basic and can be improved e.g. by detailing a selection function to be able to occasionally pick sub optimal solutions that are still deemed good enough, since evolution in practice is rarely survival of the fittest, and more often so - survival of the fit enough.

7) Literature and relevant materials

[1] Biological warfare and the coevolutionary arms race
by the Understanding Evolution team (Berkley University)

[2] Welcome to Evolution 101!
by the Understanding Evolution team (Berkley University), pages 33-36

[3] Butterflies and Plants: A Study in Coevolution, Paul R. Ehrlich and Peter H. Raven, Vol. 18, No. 4 (Dec., 1964), pp. 586-608

[4] Chess Rules Basics
https://www.chess.com/article/view/chess-rules--basics

[5] Natural Evolution Strategies
http://www.jmlr.org/papers/volume15/wierstra14a/wierstra14a.pdf

[6] Python Language and Software
https://www.python.org/