==========

TPS-GEN: Genetic Algorithm for Designing Target Prioritization Strategies

CISC455 Final Project
Joshua Chai-Tang - 20119074

==========

Section 1 - Problem Description

1.0 - Background

Like most computer science students, I enjoy playing video games in my spare time. My favourite games are ones that encourage the use of well thought out strategies. That description fits a wide variety of gaming genres, such as roleplaying games (RPG), massive multiplayer online (MMO), and real-time strategy (RTS) to name a few. Many of these games allow for strategic planning to be expressed through a combat system. The player's ability to win is often dependent on their ability to come up with strong tactical plans on the spot.

While the details of these strategies vary greatly between different games, the overall structure of combat remains relatively consistent. Resultantly, popular strategies for how best to win combat encounters tend to share similar features regardless of genre. One core aspect of combat strategy is target prioritization. Target prioritization is about choosing which enemy characters to attack and in what order they should be killed. The genetic algorithm in this report is designed to automatically generate target prioritization strategies for a variety of potential combat encounters in order to determine what the most effective strategy is.

1.1 - Combat in Gaming

The following segment is a general explanation of how combat systems work in most games, in case you're not a gamer. If you're already familiar with this concept, feel free to skim over most of this part as it won't contain much information that you don't already know.

Combat systems in different games can be vastly different from one another, but they often share similar features. Fighting generally involves at least two opposing teams, each consisting of one or more virtual characters. In most games, a team wins by killing all of the opposing team's characters first. While there are many games that fall outside this description, this algorithm is designed for combat systems that fit these requirements. Some popular examples of games that this algorithm could work for are World of Warcraft, the XCOM series, or most Final Fantasy games.

In all combat systems, characters are limited in terms of what actions they can perform. 'Attacking' is the cornerstone of combat actions; it's the general term for any kind of ability that harms an enemy. Some common examples of attacks in video games would be swinging a sword or shooting a gun. Much like performing these actions in real life, an attack must have a target. Generally the target of any attack would be an enemy character. It's up to the player to decide which enemies to target with their attacks. Therefore, knowing how to pick the right targets is essential to creating an effective combat strategy. That's what target prioritization strategy is all about: choosing the best targets for the right attacks in order to maximize the probability of victory.

As with anything that's important to gamers, this aspect of strategizing is a core talking point in heated debates across all sorts of gaming communities. Some believe that the best strategy is to prioritize killing the most powerful enemies first, while others believe that the weakest targets should be the first to go. This algorithm is designed to determine which theory, if either, is in fact more effective at winning video game battles.

Section 2 : Algorithm Design

2.0 - Simplifying the Problem

Unfortunately this problem, as I've described it so far, is simply too broad for a single algorithm to find an objective solution to. There's thousands of possible games that these strategies could be applied to, each with their own unique conditions to account for. It would be impossible to make an algorithm whose results would be useful to every game. Instead, I narrowed the search space into something more workable by making a lot of assumptions, simplifications, and abstractions of common combat system features. Resultantly this algorithm is very generic, not specifically tuned to any particular game or genre, and will thus produce generic results. While this means that the algorithm's outputs might not be directly applicable to your favourite game, the strategies it creates should still prove useful as a foundation for designing effective strategies. The entire algorithm is implemented in Python3.

2.1 - Representation

2.1.0 - The Unit

To begin, I needed a way to represent an ingame character. Most games assign two core attributes to each character: an attack damage value, and a health point value. Generally characters also have access to a wide variety of other unique abilities, special attacks, defensive perks, etc. Evidently there's no way to integrate every possible combination of unique abilities, so my algorithm instead assumes that each character consists of only two values: attack and health. The "Unit" object, defined in the representation module, represents a single virtual character. Each Unit thus has exactly two positive integer attributes: one for attack damage and one for health points. This abstraction remains useful because many common game character archetypes can be represented accurately using only two numbers. For example, the "Tank" archetype is usually a character with low damage output but who is capable of receiving lots of damage without dying. This can be represented by a Unit with a small attack attribute and a large health attribute. Conversely a "Glass Cannon" archetype, that being a character who deals lots of damage but can die easily, would have the inverse attribute distribution. With the wide range of possible attack and health value combinations, there are a lot of possibilities for testing with different types of Units.

For the rest of this report, as well as in my code, a Unit is represented graphically as <x,y> where x and y correspond to the Unit's attack and health attributes respectively. For example, <6,58> would represent a "Tank" Unit while <42,8> would be a "Glass Cannon".

2.1.1 - The Party

In order for target priority to be a factor, there needs to be multiple potential targets to choose from. This is accomplished by having combat encounters involve multiple Units, which are then grouped into teams. To keep things simple the algorithm assumes that every fight involves exactly two opposing teams, and that each team has no other objective besides killing all enemy Units. This is again an abstraction of common video game mechanics.

In my code a team is represented as a list of Unit objects, called a Party. For example, this is the representation of a possible 3 Unit Party: [<10,15>,<2,36>,<23,9>]. Each Unit in a Party is given an identifying number based on its position in the Party. In the above example, the <10,15> is called Unit0 because its index in the Party is 0. The <2,36> is thus Unit1, and the <23,9> is Unit2. These identifiers will be important to remember when discussing how Strategies are implemented in section 2.1.2.

Party dynamics will play a large role in how strategies emerge. Parties with different Unit types need different strategies, and the effectiveness of those strategies also depends on the Party composition of the opposing side. This leaves lots of potential for testing with various Party compositions.

2.1.2 - The Strategy

The representation of a Strategy is the most important component of the algorithm. Strategies are the individuals that comprise the genetic algorithm's population, defined by an object class of the same name in the representation module. It has many attributes, but the one that determines an individual's

behaviour is called ‘strat’. Therefore that’s the attribute that will be explained in depth in this section. The rest of the attributes are described in the ReadMe document attached to the source code.

A Strategy’s ‘strat’ consists of a two dimensional square array with length equal to the number of Units in the Party it’s designed for. For this report that length is called ‘N’. Each subarray contains N integers between 1 and 99. Those integers are called target priority values. Thus each subarray is a target priority value list, or TPVL. The strat is therefore a list of N TPVL’s, the purpose of which is to assign one TPVL to each Unit in the Party. A Unit’s TPVL is what determines how it picks which target to attack in combat simulations. The details of the combat simulation will be explained in section 2.3. The gist of it is that Units from both Parties take turns ‘attacking’ each other by subtracting their attack attribute from the target Unit’s health attribute. Each Unit selects its own targets independently by consulting its TPVL. Each TPVL maps exactly one target priority value to each Unit in the enemy Party. These values function as probability weights to determine how likely this Unit is to select that particular enemy Unit as its attack target.

For example, take the following Party from earlier: [<10,15>,<2,36>,<23,9>]
And take the following strat matrix: [[25,10,1], [4,4,80], [6,6,6]]
[25,10,1] is thus the TPVL assigned to <10,15> (Unit0), because that TPVL has index 0 in strat.
25 the target priority value that this Unit assigns to the enemy Unit0, 10 to their Unit1, etc.

An individual Strategy with this strat matrix would have the following behaviour:

- Unit0 prefers to target the enemy Unit0 and rarely targets their Unit2. More specifically, it has a 25/36 chance of targeting Unit0, a 10/36 chance of targeting Unit1, and a 1/36 chance of targeting Unit2.
- Unit1 prefers to target the enemy Unit2 and has no preference between Unit0 and Unit1. Specifically it has a 4/88 chance of targeting Unit0, 4/88 for Unit1, and 80/88 for Unit2
- Unit2 has no target preference at all. It has equal chances for selecting the enemy’s Unit0, Unit1 or Unit2 as its target.

Again, section 2.3 will have more details on the inner workings of the combat simulation. The important takeaway for now is that an individual’s strat matrix determines how its Units behave in combat simulations, which thus determines the phenotypes it expresses. Therefore these strat matrices are the part of an individual that is subject to recombination and mutation. More on those in sections 2.4 and 2.5.

2.2 - Initialization

Generating a new Strategy primarily consists of generating a random strat matrix, in addition to initializing a couple other record keeping attributes. Creating a strat matrix is a simple matter of generating N*N random integers between the lower and upper bounds (1 and 99) and placing them into a square 2 dimensional array. This is repeated as many times as necessary to generate the initial population. The technical details of this module can be found in the ReadMe.

2.3 - Evaluation

2.3.0 - Fitness

To determine fitness, the algorithm simulates battles between pairs of individuals. Fitness scores are awarded based on the outcomes of these battles. Winning a battle grants a large amount of fitness, while losing grants little. Additional bonuses are awarded based on the number of surviving Units at the end of the battle. The winner receives a bonus for each surviving Unit in its Party, while the loser receives a bonus for each Unit it killed. To account for the inherent randomness of combat, individuals participate in as many battles as can be feasibly simulated. Fitness is then calculated as the average fitness score the individual received across all the battles it fought. The algorithm is designed to maximize fitness. Resultantly it encourages Strategies that not only win the most, but also those that sustain minimal losses to their own Units and inflict maximum losses upon enemy Units.

An important aspect of this system is how the algorithm chooses which opponent an individual should face in battle. I developed two methods for this, each with their own advantages and issues. My code is able to switch between the two methods with the switching of a single boolean variable called “do_tournament_battles”.

2.3.1 - Method 1: Baseline Evaluation

This system works by creating a single Strategy called the ‘baseline’ at the beginning of the simulation. The baseline Strategy does not change between generations, cannot be selected for recombination, and exists outside of the population. The purpose of the baseline is to serve as a consistent opponent for all Strategies to be compared against. Under this method, individuals in the population only battle against the baseline. This ensures that all individuals are evaluated against the same criteria, which maximizes the fairness of battles. This method keeps things simple and ensures that the fitness landscape remains fixed for the whole simulation, which makes exploitation easier. After all individuals have been evaluated, a tournament system is used to select parents and survivors similar to the method used in assignment 2. The contestants with the highest fitness are selected as parents, while those with the lowest are replaced by the offspring.

The main flaw of this system is that it’s incredibly dependent on the quality of the baseline, which is handmade. If the baseline is too weak then it won’t provide adequate selection pressure, which allows for suboptimal Strategies to earn high fitness scores and rapidly take over the population. Conversely a baseline that’s too strong will crush emerging Strategies before they have the chance to develop, discouraging exploration and forcing premature convergence. In testing it proved difficult to create a baseline with the right balance of strength, so I ultimately used Method 2 to produce the results in section 3. However I still believe that this method would be useful if the correct baseline was found, hence why this section is still included in this report.

2.3.2 - Method 2: Tournament Battles

This system works by having battles occur between all potential parents during the parent/survivor selection phase. The fitness scores that result from these battles determine which individuals are selected for recombination, and which will be replaced by the offspring. This system creates a dynamic where the ‘optimal’ behaviors are dependent on the status of the population, resulting in a constantly shifting modality that evolves with every generation as Strategies learn how to counter each other. The direct competition provides strong selection pressure for potential parents, but the randomness of the tournament selection ensures that exploration is still possible.

The main flaw of this method is that it encourages individuals to be effective against the rest of their population, rather than being an objectively effective Strategy in general. Furthermore, this method vastly increases the complexity of the algorithm. Since this system requires simulating a higher quantity of battles, further multiplied by the size and quantity of tournaments, the program’s runtime becomes significantly greater than the baseline method. However this method also proved to create reliably more effective results.

2.3.3 - Battle Simulations

The battle function is the lynchpin of the algorithm. All fitness evaluations are determined by the outcomes of simulated battles, so it’s important to understand how these battles are run and exactly which assumptions they operate under. This function is intended to simulate how combat systems in video games work, albeit heavily simplified and abstracted.

Battles consist of a simulated test between two Strategies. The algorithm does not require that both Strategies use the same Party composition, but the majority of my tests were conducted under that condition. At the start of a battle all Units in either Party are set to their maximum attack and health values. After that, combat consists of a series of ‘rounds’ in which each unit makes one attack against an enemy Unit.

In the first phase of a round all Units select a target Unit in the opposite Party probabilistically according to their Strategy’s TPVL for that Unit, as described in section 2.1.2. Importantly, Units will not target ‘dead’ enemies. A Unit is considered dead if it’s health attribute is reduced to or below 0.

In the second phase of a round, Units from each Party take turns making attacks against their selected targets. An attack involves subtracting the attacking Unit’s attack value from the target Unit’s

health value. Each Unit is allowed to attack exactly once per round. Dead Units are not able to make attacks at all. After every living Unit on either side has performed an attack, the round ends. If at this point a Strategy's Party has no living Units remaining, the battle ends and that Strategy is considered the loser. Otherwise a new round begins. Rounds continue to be simulated until this condition is satisfied.

2.4 - Recombination

As mentioned earlier, strat matrices are the attribute that is used for recombination. Recombination can get tricky when the variables in question are two dimensional arrays, but I opted to pursue the route that kept things simple. Single point crossover is the method of choice, made easy by the fact that all strat matrices are guaranteed to be of equal size. Each pair of parents produces two offspring by exchanging their TPVL's, split at a randomly generated crossover point. Crossover is not performed on the TPVL's themselves. The only other thing worth noting about this method is that it's possible for the crossover point to be 0 or N, in either case resulting in a pair of offspring that are identical to their parents.

2.5 - Mutation

I didn't want mutation to be limited to only altering one target priority value or TPVL out of concern that method wouldn't scale effectively for larger values of N. Instead, the mutation method iterates through every single value in every single TPVL for the given strat matrix. At each cell it has a chance to change the value to a completely new, randomly generated one. The probability of this occurring per cell is determined by the mutation rate. Mutation is applied to every child immediately following recombination. Since every value in a strat matrix can be mutated, it's important to be careful when setting the mutation rate. The highly competitive nature of combat simulation evaluation makes this algorithm vulnerable to premature convergence, so a good mutation rate is crucial to encouraging exploration.

Section 3 : Algorithm Results

3.0 - Testing Parameters

This algorithm has an incredible amount of potential for tinkering with input parameters before each simulation. Not only does it rely on all of the standard genetic algorithm parameters such as population size and mutation rate, but the size and composition of the Party also has massive impacts on the emergent Strategies. The potential search space of this algorithm is immense, so it's important to apply some constraints to keep things workable.

I created four distinct Parties that reflected some of the most common Party compositions used in relevant games. But before getting into detail about what those Parties were, I'll first go over the other parameters I set:

Population Parameters	Fitness Score Parameters
Population Size: 100 Mutation Rate: 0.1 (10%) Tournament Size: 4 Replacement per Generation: 20 Termination Condition: 2000 Generations	Win Bonus: 5 Lose Bonus: 1 Survivor Bonus: 5 / party_size Kill Bonus: 1 / party_size

Going off what I learned from assignment 2, I decided that it wasn't beneficial to use a large population size or to simulate a lot of generations. My key takeaway from that assignment was that the best way to prevent premature convergence was to keep the population fresh by creating lots of offspring. Therefore in every generation my algorithm replaces 20 tournament losers with 20 new offspring. I also kept the mutation rate relatively high to promote exploration over exploitation. For each configuration

below, I ran at least 5 different simulations to ensure the results were consistent. The right side column results are those of the alternative algorithm, which are discussed further in section 4.

3.1 - Party Composition A : Tanks

Party: [<5,30> , <5,30> , <5,30>]

Tournament Battles Method	Particle Swarm Optimization
=== Generation 2000 === Best Individual Overall: >Strategy from Generation #1986: >Unit0: [1, 98, 12] >Unit1: [1, 79, 1] >Unit2: [1, 77, 8] >Most Recent Fitness Score: 5.88 >Lifetime Battle Count: 330 >Lifetime Win Count: 191 >Overall Winrate: 57.87%	=== Iteration 100 === FINAL SOLUTION: >Strategy from Generation #100: >Unit0: [18.33, 10.22, 1] >Unit1: [31.56, 5.60, 1] >Unit2: [22.79, 1.32, 1] >Most Recent Fitness Score: 4.40 >Lifetime Battle Count: 2000 >Lifetime Win Count: 1043 >Overall Winrate: 52.15%

3.2 - Party Composition B : Glass Cannons

Party: [<10,1> , <10,1> , <10,1> , <10,1> , <10,1>]

Tournament Battles Method	Particle Swarm Optimization
=== Generation 2000 === Best Individual Overall: >Strategy from Generation #1980: >Unit0: [98, 2, 1, 1, 2] >Unit1: [1, 93, 3, 3, 1] >Unit2: [2, 4, 98, 6, 6] >Unit3: [3, 1, 3, 98, 6] >Unit4: [1, 1, 2, 4, 87] >Most Recent Fitness Score: 5.67 >Lifetime Battle Count: 450 >Lifetime Win Count: 230 >Overall Winrate: 51.11%	=== Iteration 100 === FINAL SOLUTION: >Strategy from Generation #100: >Unit0: [6.61, 26.93, 1, 1, 4.25] >Unit1: [1.37, 2.05, 1.62, 3.63, 1.49] >Unit2: [14.17, 1, 3.13, 7.13, 1.27] >Unit3: [1.51, 1, 9.09, 1.02, 7.68] >Unit4: [1.56, 1.62, 1.25, 13.25, 1.97] >Most Recent Fitness Score: 3.62 >Lifetime Battle Count: 2000 >Lifetime Win Count: 809 >Overall Winrate: 40.45%

3.3 - Party Composition C : Boss Battle

Party: [<20,100> , <5,10> , <5,10> , <5,10> , <5,10>]

Tournament Battles Method	Particle Swarm Optimization
=== Generation 2000 === Best Individual Overall: >Strategy from Generation #1983: >Unit0: [1, 71, 53, 14, 1] >Unit1: [3, 2, 5, 27, 97] >Unit2: [2, 1, 4, 11, 48]	=== Iteration 100 === FINAL SOLUTION: >Strategy from Generation #100: >Unit0: [1.02, 7.08, 10.43, 13.48, 1.0] >Unit1: [4.75, 18.72, 1.06, 1, 4.64] >Unit2: [1, 3.77, 1, 1, 21.59]

>Unit3: [2, 5, 2, 8, 86] >Unit4: [9, 3, 3, 95, 23] >Most Recent Fitness Score: 5.18 >Lifetime Battle Count: 450 >Lifetime Win Count: 245 >Overall Winrate: 54.44%	>Unit3: [1, 16.62, 9.58, 3, 9.32] >Unit4: [3.82, 1, 12.48, 1.03, 10.49] >Most Recent Fitness Score: 2.90 >Lifetime Battle Count: 2000 >Lifetime Win Count: 520 >Overall Winrate: 26.00%

3.4 - Party Composition D : Balanced

Party: [<5,45> , <25,25> , <30,20> , <15,35>]

Tournament Battles Method	Particle Swarm Optimization
=== Generation 2000 === Best Individual Overall: >Strategy from Generation #1988: >Unit0: [4, 88, 63, 29] >Unit1: [1, 81, 1, 2] >Unit2: [1, 2, 70, 5] >Unit3: [1, 58, 96, 46] >Most Recent Fitness Score: 6.29 >Lifetime Battle Count: 240 >Lifetime Win Count: 129 >Overall Winrate: 53.75%	=== Iteration 100 === FINAL SOLUTION: >Strategy from Generation #100: >Unit0: [7.55, 2.27, 15.75, 12.89] >Unit1: [1, 7.54, 21.76, 1] >Unit2: [8.03, 1.0, 17.08, 8.94] >Unit3: [1.0, 14.29, 1.0, 3.12] >Most Recent Fitness Score: 3.30 >Lifetime Battle Count: 2000 >Lifetime Win Count: 622 >Overall Winrate: 31.1%

Section 4 : Comparison to Alternatives

4.0 - Particle Swarm Optimization

For the alternative algorithm, I chose to adapt a particle swarm optimization module by Nathan Rooy [2]. Particle swarm optimizations are relatively simple and often less computationally intensive than genetic algorithms, so they're a good candidate for determining if effective Strategies can be produced using fewer resources. There was no easy way to have the particles in Rooy's PSO module compete directly with each other, so the particle fitness evaluation function used was the baseline method described in section 2.3.1. I ran multiple PSO simulations for every Party composition, each using a population of 25 particles run for 100 iterations. The best PSO Strategies were then compared to the best GA Strategies by simulating battles between them. Note that while the fitness scores in the left columns of section 3 are from the GA's individuals competing with their own population, the fitness scores in the right columns are from the PSO's best Strategies battling directly with the Strategies in their respective left columns.

The PSO Strategies were remarkably similar to those of the GA in most cases, especially for composition A. However these Strategies were also less optimized, often losing the majority of battles against the best GA Strategies. Overall the GA produced higher quality solutions. It should be noted that the PSO algorithm likely suffered from the same flaws in the baseline evaluation method that made me abandon it for the GA. The PSO also took significantly less time to simulate, often completing all 100 iterations in fewer than 2 minutes. Compared to the average 15-20 minute runtime of the GA, that is a significant runtime improvement in exchange for Strategies that are still somewhat effective. The PSO algorithm has demonstrated a promising start, and it certainly holds the potential to be equal or superior to the GA if given adequate development time.

Section 5 : Discussion

5.0 - Results by Party Type

5.0.1 - Composition A: Tanks

This Party consisted entirely of low attack, high health units. The dominant Strategies produced always focused on targeting the same Unit first, though which specific Unit got chosen varied between simulations. The secondary target was never as focused. The two smaller target priority values were usually of similar or equal value. It appears that once a numbers advantage has been achieved over the enemy Party, the target priority Strategy doesn't particularly matter.

5.0.2 - Composition B : Glass Cannons

This Party consisted entirely of high attack, low health units that could easily kill each other with a single attack. The Strategies that succeeded here were essentially the inverse of those from Party A; Since targeting the same Unit provided no advantage towards killing it faster, the best Strategies spread out their attacks evenly. Each Unit had its own preferred target with a priority value in the high 90's, while all other targets were assigned values in the single digits. Again, the best Strategy was to kill enemies quickly and gain a numbers advantage early.

5.0.3 - Composition C: Boss Battle

This composition features one extremely powerful Unit with high attack and health, surrounded by numerous weaker Units with much smaller values. This composition emulates the traditional 'boss battle' archetype that is common in all sorts of gaming genres, with the one stronger Unit representing the 'Boss' and many weaker units representing its minions. The dominant Strategies here focused on killing those minions first. All Units assigned the Boss incredibly low priority, targeting it last. Another interesting property was the way that different Units chose to spread out their target priorities. The population was able to recognize that there was no benefit in having the minions attack the same target as the Boss, as the Boss was already capable of killing enemy Units in a single attack. Thus it developed a Strategy that split target priorities evenly between the Boss Unit and its collection of weaker Units.

5.0.4 - Composition D : Balanced

This composition features a diverse combination of Units with a wide variety of attack and health distributions. Notably the attack and health values of each Unit sums to 50, hence the name 'balanced' because all Units have comparable attributes. This composition more closely emulates what a group of Player characters would look like, since Player characters generally have differing strengths and weaknesses but relatively similar overall power. Therefore this composition mostly simulates "PvP" (Player vs Player) battles.

The dominant Strategies prioritized targets with low health values, whilst largely ignoring those with low attack values. Unit0, having the most health and least attack, consistently received the lowest target priority value. Unit1 & Unit2, having relatively little health, generally received the highest target priority values. Consistent with the results of the other party compositions, the best Strategies focused on reducing the total quantity of enemy units quickly.

5.1 - Conclusions

The results from Party compositions C & D indicate that the best target priority strategy is indeed to focus on killing the weakest enemies first. The dominant Strategies from both tests assigned highest target priority to Units with the least health while giving lowest priority to Units with the most health, regardless of how large their attack value was. All dominant Strategies produced, regardless of Party composition, strongly valued reducing the quantity of enemy Units as quickly as possible. Thus the answer to the target priority debate is clear; In general, the optimal Strategy is to focus on killing the weakest targets first and dealing with the most dangerous targets last.

However, due to the limited number of Party compositions tested I cannot confidently assert that these results are universally true. This algorithm cannot begin to account for every possible game or

scenario. In the future I'd like to test this algorithm with more Party types, especially with asymmetric Parties, to see how different dynamics affect the outcome. It would also be interesting to pursue adding further complexities to combat simulations, such as granting Units special abilities based on common gaming role archetypes, to see how that alters the results. While there's still an incredible amount of potential for additional testing with this algorithm, I also believe that the current results are generically applicable to most games.

Section 6 : References

[1] Github Repository for this Algorithm, by Joshua Chai-Tang
https://github.com/chai-tang/TPS-GEN

[2] Particle Swarm Optimization, by Nathan Rooy
https://github.com/nathanrooy/particle-swarm-optimization