2022

Adaptive NPC Behavior In Maze Chase Game Using Genetic Algorithms

Karen Suzue
The College of Wooster, ksuzue22@wooster.edu

Adaptive NPC Behavior in Maze Chase Game Using Genetic Algorithms

Independent Study Thesis

Presented in Partial Fulfillment of the Requirements for the Degree Bachelor of Arts in Computer Science in the Department of Mathematical Computer Sciences at The College of Wooster

by
Karen Suzue
The College of Wooster
2022

Advised by:

Drew Guarnera (Computer Science)

THE COLLEGE OF
WOOSTER

Abstract

Difficulty adjustment, although an important aspect of game development, is a tedious and costly process that produces uncertain results due to the wide range of skill among players. Most of the current approaches to difficulty adjustment in video games include fixed incremental difficulty curves, data generalization, and extensive manual playtesting. In this study, we explore genetic algorithms as an alternative approach to difficulty balancing with a focus on adjusting NPC behavior. A maze-chase game is built for demonstration purposes. Additionally, a genetic algorithm based on previous theoretical techniques is developed for the game.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Game balancing is a process that involves fine-tuning a game to promote fairness and maximize player satisfaction. In the field of game design, game balance is a crucial part of a game's commercial success. An imbalanced game, whether that is through an overpowered weapon or a lack of diverse paths to victory, can greatly hinder the player's enjoyment and thus engagement. Despite its importance, game balancing is a tedious and difficult task due to the diverse range in skill and preferences among players, often requiring extensive manual trial and error testing or other forms of qualitative research [7].

One of the common types of game balancing is diffculty adjustment, which is the focus of this Independent Study. As of now, no universal model exists for any form of difficulty adjustment. Most games rely on fixed incremental difficulty curves (e.g., "easy", "medium", "hard" modes) otherwise known as static difficulty adjustment [17]. This means that the level of difficulty, once chosen, does not often shift over the course of gameplay, or can only be changed according to a limited set of options. This type of difficulty adjustment is popular in game design as it provides an easier solution to maintaining difficulty balance and thus player engagement. However, with so many factors of player engagement being highly subjective values, it is challenging to pinpoint the right level(s) of difficulty to implement in a game. Furthermore, these difficulty levels are often derived from data generalizations or

pure guesswork - both of which do not guarantee that everyone will have the same desired experience while playing them.

To mitigate these issues, much research has been conducted on dynamic difficulty scaling methods which automatically change a game depending on the player's progress or performance. There are many benefits to using dynamic difficulty adjustment, namely, less manual testing involved during game development and ensuring that the different needs of most, if not all players, will be met. While there are many different approaches to developing a dynamic difficulty adjustment method, utilizing machine learning algorithms have shown promising results thus far. One of such algorithms is genetic algorithms, a group of heuristic, unsupervised machine learning algorithms that uses principles inspired by natural selection and evolution [12]. In genetic algorithms, candidate solutions mate and compete against each other until convergence occurs [12]. Created to solve nonlinear optimization or search problems for which little information is known, genetic algorithms do not depend on training sets of input and output data [12]. Thus, this type of algorithm is a good choice for accommodating uncertainties such as establishing the right difficulty for unknown player skill levels.

This Independent Study explores the use of genetic algorithms as an alternative approach to solving the problem of balancing difficulty in video games. The genetic algorithm implemented focuses on evolving non-playable character (NPC) behavior. Although different factors in gameplay could be manipulated for the purpose of difficulty balancing, NPC behavior provides a large portion of the players' interactions and experiences in many genres. Thus, NPCs can determine not only a significant part of gameplay but also difficulty. To aid our investigation into the use of genetic algorithms for dynamic NPC behavior, a Pac-Man inspired maze chase game and genetic algorithm are developed for demonstrational purposes. This type of game was chosen for its reliance on using NPC behaviors as its main

game mechanic. A genetic algorithm is implemented to run alongside gameplay, allowing the player to play against enemy characters to collect performance data.

CHAPTER 2

Related Works

2.1 Game Design Theory

The study of game design is mainly concerned with maximizing player experience through exploring and leveraging human behaviors. In addition to genetic algorithms, this Independent Study focuses on two concepts belonging to this area of study, which are game balance and theory of flow. Explanations for such concepts are provided in this section.

2.1.1 Theory of Flow

The theory of flow is a well-established theory commonly cited in the area of game design to help maximize player enjoyment [2]. The theory defines flow, a mental state marked by complete absorption and investment while carrying out a certain activity [5]. This state of deep immersion is often found in (but not limited to) recreational, intrinsically rewarding activities known as play in the field of psychology [5]. As defined by Csíkszentmihályi, the state of flow is identified by six key elements [5] :

- The merging of action and awareness, or "merged awareness", which entails a sense of oneness with the task at hand. While performing tasks in a non-flow

state often includes an awareness of limitations and reflective thought, the flow state makes tasks feel automatic and effortless to the individual. For this merge to happen, the task must be within one's capabilities.

- The centering of attention which involves limiting one's stimulus field. Any stimuli irrelevant to the current task is ignored. In many cases, this shifting of focus arises from rules and sources of motivation which help define what relevant stimuli are in a given situation.

- Loss of self-consciousness. The state of flow provides a temporary environment free from the constraints or consequences of everyday life and thus the awareness of self that arises from it.

- Sense of control over one's actions or their environment derived from the absence of worry. This makes performing tasks under the state of flow feel manageable and low-stake. Once again, this requires the task to be feasible for the individual. Additionally, this is connected to the loss of reflective self-consciousness which removes the inner critic from one's thought processes.

- Unambiguous demands for action and feedback. Goals and means are apparent and non-contradictory. Simple rules along with feasible tasks often make this aspect easy to achieve.

- Its autotelic nature. Flow is intrinsically rewarding and motivating. In this state, process often takes precedence over goals.

For all activities, flow is regarded as the state of optimal experience as it maximizes enjoyment and satisfaction [2]. For this reason, it is crucial for games to induce and maintain a state of flow. In order to achieve this, conditions that give rise to this state must be identified. According to Csíkszentmihályi, three conditions must exist for flow to be induced [5]. The first condition requires a task to have clear

sets of goals. As flow is autotelic, goals in this state are not seen as ends but rather a means to provide structure and direction to a task [5]. Another key condition for flow to arise is clear and immediate feedback. In everyday life, decision making involves constantly negotiating between changing and often contradictory demands, making it difficult to achieve or maintain flow. The purpose of having clear and immediate feedback is similar to that of the first condition, that is to provide direction for the task at hand [5]. Additionally, clear and immediate feedback leaves the individual with little doubts regarding their next action. Feedback serves to inform the individual how they are progressing in an activity, which in turn dictates how they will adjust or maintain their current actions [5]. Csíkszentmihályi states that feedback is inherently negative when dealing with challenging tasks [5]. This is seemingly detrimental to maintaining flow, which is meant to be a positive experience. Despite this, it is still possible for flow to occur due to a third condition - a balance or match between perceived challenge and ability. Flow is induced when one sees the challenge as feasible or not too difficult for their skill level [5]. As seen in Fig. 2.1, the model of flow predicts that flow can occur at high levels of challenge or demands for action, as long as there is also a high level of skill to match.

As demonstrated by the model of flow depicted in Fig. 2.1, this balance is very fragile, and thus flow can not be easily maintained for extended periods of time. An individual who is unable to cope with the current challenges or demands for action becomes anxious. Meanwhile, when their capabilities exceed the challenge at hand, boredom occurs. Additionally, if their skills overwhelmingly exceed the challenge, anxiety overcomes the state of flow once more [5].

It is important to emphasize the subjective nature of the third condition. Csíkszentmihályi states that flow depends entirely on one's perception of their own skills and the challenges at hand. Thus, it is entirely possible for someone to move between states of anxiety, boredom, and flow while performing the same task [5].

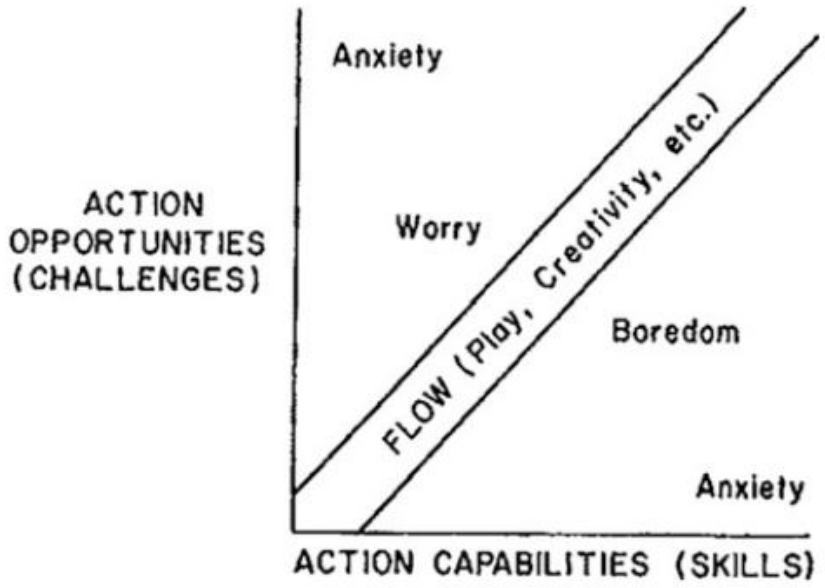

Figure 2.1: Csíkszentmihályi's Model of Flow [5]

Furthermore, everyone experiences flow differently. Some people are more prone to flow and can sustain it, while others can not. The frequency and quality of flow is experienced at varying levels from one person to another [2].

Recent research shows an apathy state that is experienced when both skill and challenge, although perfectly matched and within the flow channel, are low [2]. Cruz and Uresti proposes a new model that extends on the original model to reflect these findings and provide a more detailed description of an individual's emotional state. This eight channel model, as seen in Fig. 2.2, is centered around the concept of a "subject mean", which emphasizes the individualized nature of the flow experience. "Subject mean" refers to the individual's ability. According to this model, an individual achieves the flow state when both challenge and skill, while being equally matched, are perceived as above their subject mean. Additionally, new states such as arousal, control, and relaxation are included. States previously seen in the original model such as worry, anxiety, or boredom are given new placements on the graph [2].

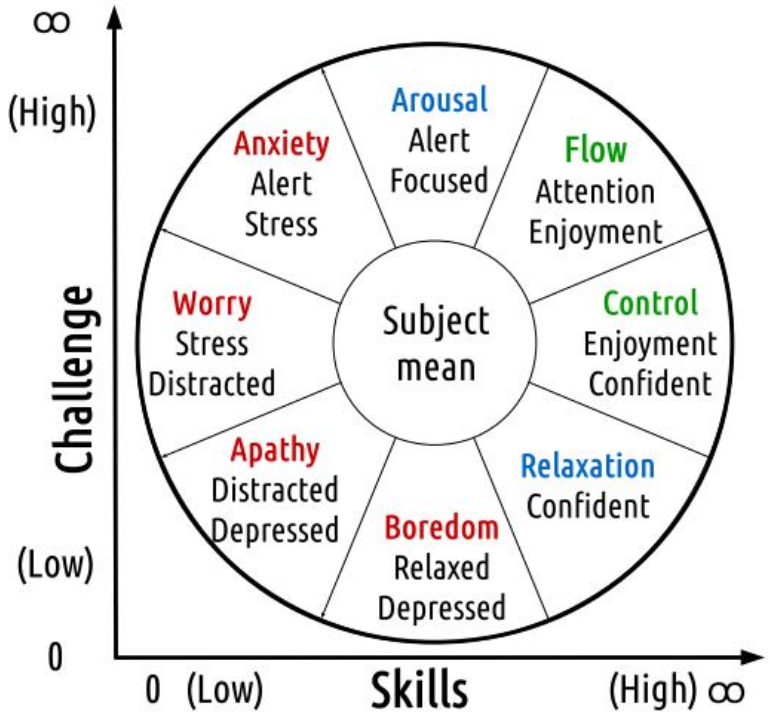

Figure 2.2: Cruz's and Uresti's Eight Channel Model of Flow [2]

Both models imply that it is difficult for games with static difficulty and mechanics to maintain a state of flow. To apply the model in such games, one has to assume that players will perceive skills and challenges objectively. Due to the model being based on entirely subjective measures of challenge and skill, it is not a reliable predictor of when a player will achieve flow, even with objectively feasible challenges. Dynamic game mechanics can, however, avoid this problem and foster or maintain the player's flow. Additionally, by specifically utilizing artificial intelligence techniques, one can completely tailor the elements of the game according to the player's needs, therefore bypassing the subjectivity of their experience and perceived skill.

2.1.2 Game Balancing

Game balancing, which involves fine-tuning various in-game factors to match with the capabilities and demands of the player, is a crucial aspect of game design. This is because having good game balance fosters flow and maximizes player enjoyment or satisfaction. While there are many factors involved in creating a balanced game, one of the most common types of game balancing is difficulty adjustment. This often involves ensuring a balance between challenge and what is within the player's skill level - the very basis of a flow state [11].

Static difficulty adjustment is currently the most widely employed form of difficulty balancing in video games. In many games, players are given a fixed range of predefined difficulty modes to choose from. Such difficulty modes are not specifically tailored according to the player's needs; they are general speculations regarding what the majority of players may be capable of or might find interesting. It is difficult for games with predefined difficulty levels to consistently optimize player satisfaction as this approach does not take into account the wide range of skills and preferences players have. Furthermore, players' capacity to improve and adapt over time varies at different rates. Thus, the progression in difficulty may not line up with the player's learning. Similarly, a player's learning strategy may be completely different from the developer's expectations [1].

Currently, much of difficulty balancing is done manually through repeated trial-and-error testing [7]. This process, known as playtesting, is not only time consuming but also costly. Furthermore, determining what adjustments to make based on feedback from playtesting is difficult, and subtle changes can drastically and unpredictably affect the rest of the game. An example of this occured during the development of Halo 3, in which a sniper rifle weapon was found to be overpowered. Many small adjustments were made including clip size, time between shots, the maximum amount of ammo, as well as reload time, with each change involving

extensive playtesting. As such, implementing static difficulty is largely dependent on costly but fuzzy answers and intuition, both of which are not very reliable or effective [7].

As games vary greatly in terms of mechanics and design, no mathematical formula or universal model exists for the purpose of game balancing [7]. An alternative approach would be dynamic difficulty adjustment, which adapts various in-game factors to the player's skill level while maintaining enough difficulty to keep the game interesting. Using adaptive mechanisms allows the difficulty adjustment process to become more fine-grained and in tune with the player's progression or learning strategies. While widely different approaches have been proposed and tested for dynamic difficulty adjustment, such as reinforcement learning or genetic algorithms - the latter of which is the focus of this Independent Study - Andrade et al. identifies three basic requirements that all methods must satisfy. Firstly, such methods must be able to quickly identify and adapt to the player's skill level. Secondly, the method must track the player's progression closely and swiftly. Thirdly, the method must provide believable and seamless changes. For example, the game's enemy should not have to perform several self-defeating actions before coming up with a suitable resolution. For dynamic difficulty adjustment, timeliness is a big factor. Any delays or misalignment in progression will remove the player from the flow state [1].

As mentioned previously, there are various approaches to implementing dynamic difficulty adjustment. An approach is to have an implicit or explicit measure to keep track of the difficulty level the player is facing. This measure could be as function that maps various in game factors to a value that describes how difficult the game feels to the player. For example, a measure such as rate of successful hits or game score can be used to interpret the game's difficulty. An issue that arises from solely using this approach is the mismatch between perceived and objective

difficulty. Thus, this may not be the best method for optimizing flow. Another approach to dynamic difficulty adjustment is to control the game environment, such as giving players more weapons or increasing their life point recovery rate. A problem with this approach is the number of rules that have to be manually implemented, which is not only time-consuming and difficult to maintain but also error-prone. Furthermore, there is a limit to the game's adaptive capabilities using this method [1].

A more intuitive and innovative solution would be to use machine learning to adapt game agents or environment to the player's skills [1]. Should this be successful, much of the unpredictable problems and time-consuming manual work can be avoided. Additionally, most of the issues that arise from the approaches mentioned above can be solved with artificial intelligence. In exchange for higher adaptability, however, machine learning methods often deal with problems regarding timeliness as convergence can take a long time to achieve. This would lead to long waiting times on the player's end which disrupts their flow. The following section discusses the different ways in which certain genetic algorithms mitigate this problem.

2.2 Genetic Algorithms

This Independent Study aims to use genetic algorithms as an alternative approach to game balancing. Genetic algorithms provide an alternative heuristic approach to Artificial Neural Network (ANN) training that is based on natural selection and biological evolution [12]. Additionally, it is an unsupervised machine learning method which doesn't rely on training sets of input-output data [12]. For this Independent Study, genetic algorithms were chosen over other machine learning methods due to the number of existing research conducted on closely related topics. Video games are popular test beds for genetic algorithms, which leads to a

prevalence of game-related research for this type of machine learning algorithm. Additionally, game balance and maximizing player enjoyment are both difficult to measure or predict, even with large sample data. Problems that involve highly probabilistic interactions such as these are suitable for genetic algorithms, which do not depend on example pairs of input-output data that are typical of supervised learning algorithms.

2.2.1 Basic Concepts

All genetic algorithms search through a space of potential solutions or hypotheses in order to find the best fit hypothesis for a given optimization problem. Such space is known as the population. In each iteration of the algorithm, the population is updated to form the current generation. During this process, hypotheses in the current generation are evaluated based on a fitness score that determines their accuracy compared to given training data or according to a set of constraints. Individuals with lower fitness are then removed to make space for better hypotheses in the next generation. While some hypotheses are kept intact, others are used to produce new hypotheses for the next generation. For this to occur, all genetic algorithms undergo crossover and mutation operations similar to that of sexual reproduction and evolution in nature in order to produce new solutions [12].

Algorithm 1 displays pseudocode for a prototypical genetic algorithm. In this algorithm, the Fitness parameter represents a fitness function that returns a numerical score, while FitnessThreshold represents a chosen maximum fitness value that acts as the terminating condition for the loop. Additionally, p represents the fixed number of hypotheses in a population throughout all iterations, r represents a fraction of the population to be replaced by the crossover operation, and m represents the percentage of hypotheses to be mutated with uniform probability [12].

Algorithm 1 Prototypical Genetic Algorithm

function GA(Fitness, FitnessThreshold, p, r, m)
Initialize population with p hypotheses
Evaluate each hypothesis h in P against Fitness function
while maximum fitness of a hypothesis in the population is less than FitnessThreshold **do**
Create new generation P_s
Probabilistically select $(1 - r)p$ hypotheses from P to add to P_s
Perform crossover on $\frac{r \cdot p}{2}$ pairs of most fit hypotheses. Produce two offsprings for each pair, add all to P_s.
Perform mutation on m percent of hypotheses in P_s chosen with uniform probability
Update current population to P_s
Evaluate each h in P_s against Fitness function
end while
Return hypothesis with highest fitness

The first step in the algorithm is to initialize a population P with p randomly generated hypotheses. Through each iteration, every hypothesis is evaluated through the Fitness function. If the largest fitness value among all hypotheses is smaller than the fitness threshold, the algorithm creates a new generation, which is denoted by P_s, the successor population. To update the current population, a group of hypotheses is chosen to be added to the next generation. This is done based on probability, such as that calculated by Equation 2.1 below [12]:

$$Pr(h_i) = \frac{Fitness(h_i)}{\sum_{j=1}^{p} Fitness(h_j)} \tag{2.1}$$

This equation describes the probability of being chosen for a hypothesis h_i. In most cases, this probability is proportional to the fitness value of h_i and inversely proportional to the fitness values of other competing members in the population. For this prototypical algorithm, the probability that h_i is chosen is determined by dividing its fitness value by the sum of all fitness values from the other hypotheses. Thus, a fitness value that is higher in comparison to that of other members in the population increases the chances of hypothesis h_i to be chosen.

The next step of the genetic algorithm is the crossover operation which essentially combines parts of two parent hypotheses to produce a new child hypothesis. Before crossover can occur, the algorithm first selects $\frac{r \cdot p}{2}$ pairs of the most fit hypotheses. Then, for each pair of hypotheses the crossover operator is applied. The inner workings of crossover operators differ between each genetic algorithm and largely depend on the genome representation or encoding chosen for such algorithm. Thus the specifics of the crossover process will be discussed at length in the following sections.

In addition to crossover, the algorithm also goes through a mutation operator. The mutation step of the algorithm forms new hypotheses by making random changes to existing hypotheses. These existing hypotheses, which together make up to m percent of the members of the population, are also randomly chosen. Once again, the specifics of this process differs between each genetic algorithm, and will be explained in the following sections.

2.2.1.1 Genetic Representation

In genetic algorithms, hypotheses, also referred to as genomes, are typically represented using fixed-length bit strings. Such bit strings are divided into specific substrings that are used to represent attributes, otherwise known as genes. Within a substring, each bit position represents a possible value it can take on and thus a constraint or rule, with 1 enabling the constraint ('true' or 'yes') while 0 disables it ('false' or 'no') [12].

To demonstrate how this representation is applied, consider the problem of encoding if-then rules. These rules, as hypotheses, act like boolean-valued functions defined over a set of data, i.e. for any given instance the hypothesis returns 0 or 1 depending on whether the instance satisfies its constraints. For this example, assume that a rule precondition consists of a conjunction of two different attributes -

Outlook and *Temperature* [12]. For the attribute *Outlook*, there exists three possible values - *Sunny*, *Overcast*, and *Rain*. This can be represented using a three bit string, where each bit position represents a value available for the attribute. Thus, a bit string of 010 represents the constraint of $Outlook = Overcast$, while 111 represents the most general constraint, namely $Outlook = Sunny \vee Overcast \vee Rain$. Similarly, for the attribute *Temperature*, there exists four values that can be encoded into a string of three bits. In this case, each bit represents the values *Hot*, *Cold*, *Warm*, and *Cool*. Together, these bit strings would form a larger string of length seven that make up the precondition statement. Suppose that the postcondition is *PlayTennis*, which takes on values of *Yes* and *No*, an extra bit can be added which results in an eight bit string. Thus, a rule such as

$$IF\ Outlook = Sunny \wedge Temperature = Cool\ THEN\ PlayTennis = Yes \tag{2.2}$$

can be represented by the bit string

$$10000011 \tag{2.3}$$

As this is a fixed-length string representation, attributes in a hypothesis can be identified easily [12].

2.2.1.2 Operators

In genetic algorithms, operators are used to generate new solutions from pre-existing solutions. Two types of operators exist in genetic algorithms: crossover and mutation. While crossover serves to retain attributes from previous hypotheses with high fitness, mutation is used to introduce new attributes or genes into the population. Both operators require solutions to be encoded in ways that make it

possible for genes or attributes to be extracted and identified, such as in the case of using bit encoding. The ability to easily manipulate attributes is one of the main advantages of using bit-strings. With fixed-length bit-string representation, one can perform the crossover or mutation operations by simply modifying bits in the string [12].

All crossover operators dealing with bit-string representation involve copying selected bits or substrings from each parent hypothesis through a "crossover mask". This is illustrated in Figure 2.3 below:

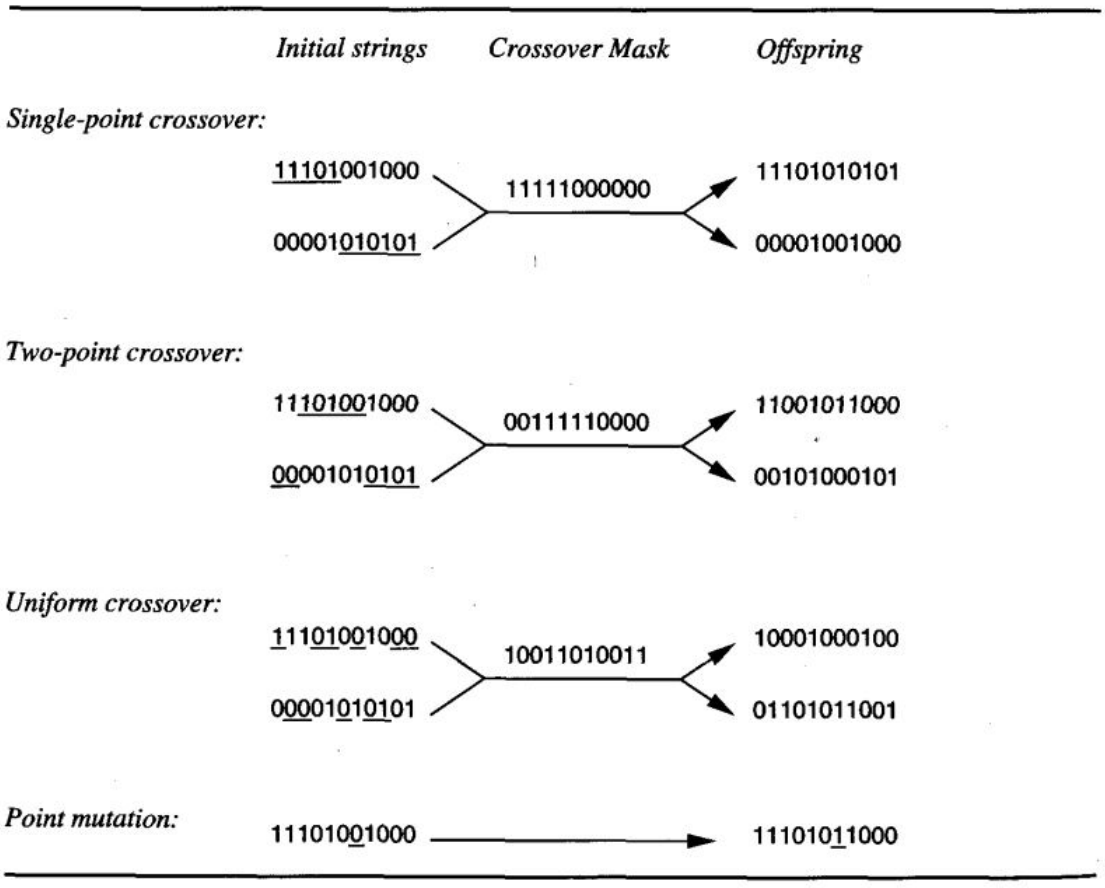

Figure 2.3: Common operators for bit string representation (including mutation operator) [12]

Figure 2.3 displays the common types of crossover operators for genetic algorithms with bit-string representation. There are many different ways of performing

crossovers, each with different sets of strengths and weaknesses. The most simple type of operator, the single-point crossover, works by selecting a single pivot point for bits to be exchanged using a crossover mask. For a hypothesis with n bits, the single-point crossover operator chooses the first m bits from one parent and the remaining $n - m$ bits from the other parent. The offspring would then have its first n bits from the first parent, and $n - m$ bits from the second parent. In contrast, two-point crossover works by selecting two crossover points and exchanging the bit substrings in between or outside of those two points [9].

While single and two-point crossover operators are relatively simple to implement, they provide very limited options for information exchange. For two parent hypotheses of length n, the single-point operator only allows for $2n$ exchanges. As for two-point operator, there are only $n^2 - n$ outcomes. A more effective way would be to perform uniform crossover, which selects bits instead of segments. Such bits are chosen randomly and independent of other bits in the string. The crossover mask for this type of operator is created by generating a random bit string with each bit chosen at random and uniformly - that is each bit has the same probability of being chosen. This type of operator allows for a significantly higher number of 2^n outcomes and is suggested through research to be more effective than single point and two point crossover methods [9].

Mutation, unlike crossover, only requires a single parent. In the mutation process for bit string representation, a single bit or more are randomly selected and modified as seen in Fig. 2.3. Mutation is often performed after the crossover process [12] and is useful for creating variation or recovering lost attributes in the population. It is intended to be most effective after the algorithm has reached a local minima, converging to a solution that is good but not the best [9].

2.2.2 rtNEAT

rtNEAT is the real-time implementation of the NeuroEvolution of Augmenting Topologies (NEAT) algorithm. Its purpose in the original study conducted by Stanley and Miikkulainen is to train agents that could respond and adapt to the player's actions dynamically [16]. In the study, rtNEAT served as the core mechanic for the experimental game NeuroEvolving Robotic Operatives (NERO), which was solely developed for this research. The premise of NERO revolves around the player training a team of agents to battle against other teams in the game. The training is done in real time with the help of the rtNEAT algorithm. Over time, as the agents gain experience and more training is given, their behavior complexifies and adapts to the player's tactics [16].

At the fundamental level, rtNEAT is indistinguishable from NEAT and retains all features and operators of the original algorithm. Unlike many genetic algorithms, rtNEAT or NEAT uses neural networks instead of bit strings for hypothesis representation. NEAT evolves structures and weights of neural networks by repeatedly performing crossover and mutation operators on the best hypotheses in the current population or generation. The algorithm begins with simple, randomized neural network structures and evolves hypotheses on its own without the need for a predetermined starting topology. By doing so it is able to efficiently converge to the simplest solution through incremental structural changes. Additionally, this allows NEAT to minimize the search space much faster with less evaluations during training compared to other Neuroevolution (NE) methods [15].

2.2.2.1 Genetic Representation

NEAT and rtNEAT evolves hypotheses or genomes in the form of neural network structures and sets of weights and genes. According to Stanley and Miikkulainen, there is a distinction to be made between these two forms of representation [16].

The set of weights and genes are described as genotypes whereas neural networks that result from the mappings carried by such genotypes are called phenotypes (Fig. 2.4). Both are representations of a single genome [16].

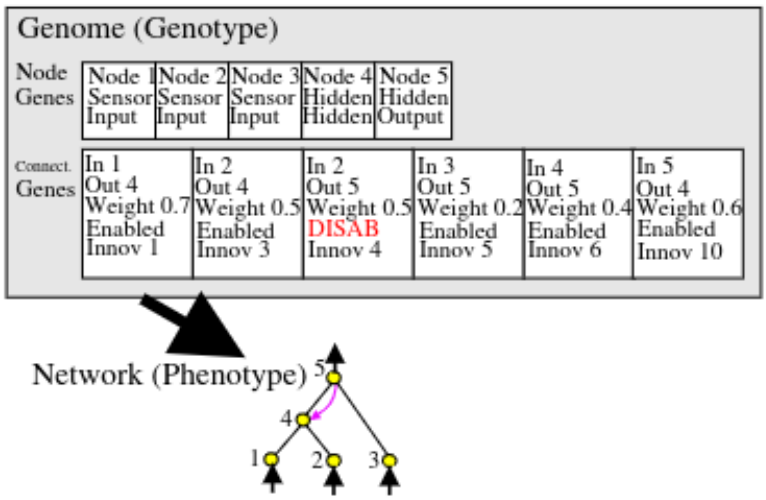

Figure 2.4: Mapping of genotype into phenotype [16]

In NEAT, each hypothesis or genome contains two types of genes: connection genes and node genes. Node genes are neurons in the network, whereas connection genes conceptually refer to the connection between two neurons. To be specific, connection genes are sets of a weight value and two connected node genes - the in-node and out-node, with in-nodes situating in the lower levels of the neural network as shown through the direction of arrows illustrated in Fig. 2.4. The weight value of the gene determines how much influence it has over the final solution. Additionally, a connection gene set also contains an enable bit as well as an innovation number, both of which play an essential part in the crossover and mutation process. An enable bit keeps track of whether or not the connection gene is active, while the innovation number keeps track of a connection gene's historical origin and identifies matching genes for the crossover process [14].

2.2.2.2 Initial Population

Genetic algorithms that evolve neural networks often start with a completely randomized set of topologies [16]. This strategy often takes many generations for the algorithm to find a solution [16]. NEAT, however, begins with an initial uniform population of network with no hidden genomes and random weight values [16]. This allows the algorithm to begin minimally and have networks grow only out of necessity. New structures are introduced to the population incrementally through mutations, and survive only when they are proven to be useful over time. An advantage of this approach is that NEAT requires significantly less evaluations than that of the norm and efficiently minimizes the search space. The resulting solution is thus minimal and optimal [16].

2.2.2.3 Operators

In both NEAT and rtNEAT, crossover is done by matching the connection genes of two parent genomes according to their innovation numbers. As illustrated in Fig. 2.5, NEAT's crossover process begins by aligning the genes according to their innovation numbers. Genes that don't have a match are called disjoint or excess genes, which are topologies or attributes seen in only one genome and not the other [14]. In most cases where two genomes have varying levels of fitness, genes with matching innovation numbers are randomly inherited while excess genes are chosen from the fitter parent. In the case where both parents are of equal fitness, however, excess genes are also randomly inherited, as shown in Fig. 2.5 [16].

Unlike what occurs in the mutation process for bitstring representation, mutation for NEAT revolves around manipulating the topology of neural networks. This is done by adding node or connection genes. Both types of genes are involved in the mutation process, thus two types of mutation can occur: add connection mutation and add node mutation. As shown in Fig. 2.6, add connection mutation

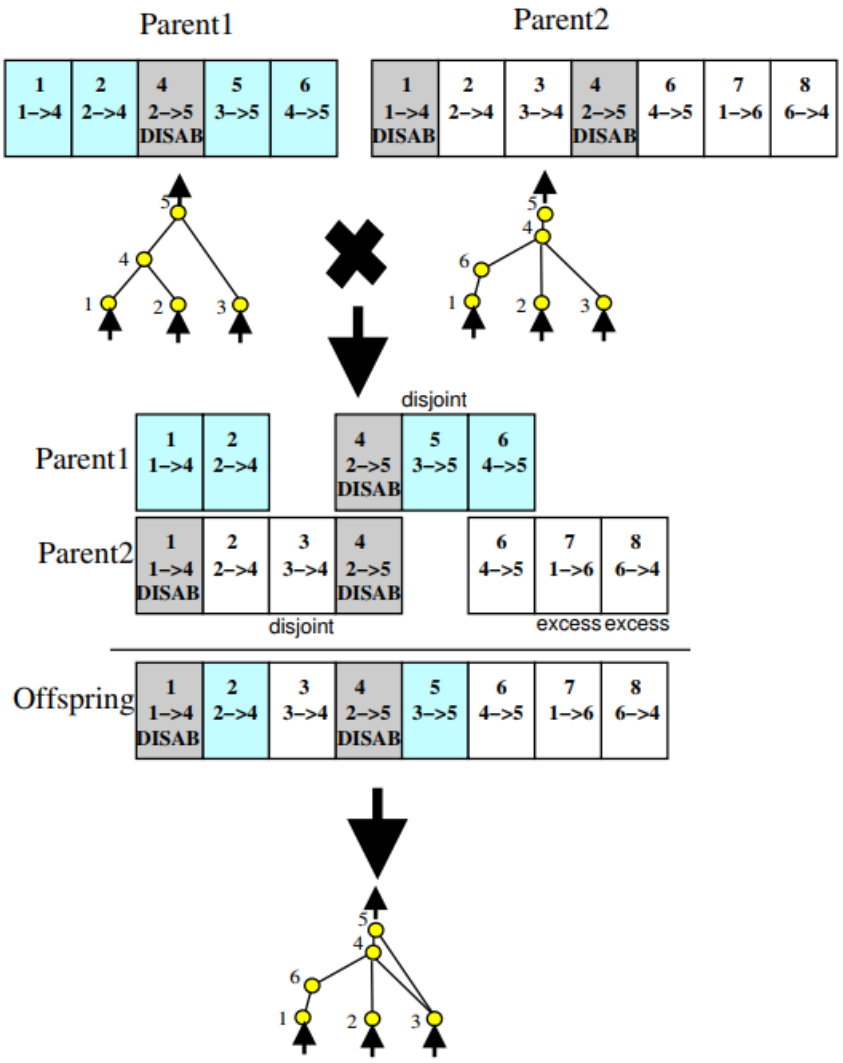

Figure 2.5: NEAT's Crossover Process. The numbers on top of each box represents the innovation number of each gene [16]

involves inserting a new connection between two pre-existing but unconnected nodes in the neural network. This new connection is also assigned a random weight value. Meanwhile, add node mutation involves inserting a new node in between two previously connected nodes. This leads the existing connection to disable and results in two new connection genes. The weight of the original connection is then transferred to the second half of the split that leads out from the new node while the first half leading into the node receives a weight value of 1 [14].

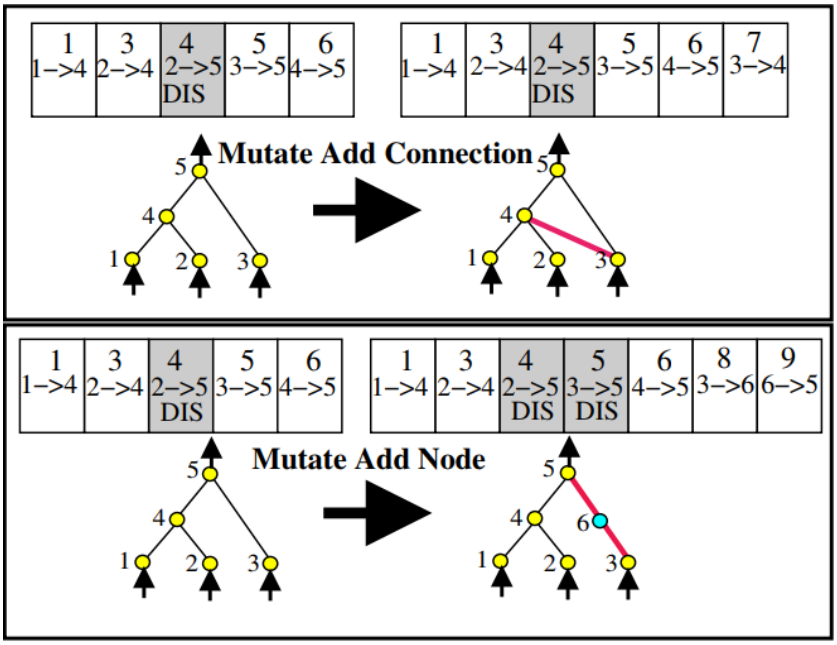

Figure 2.6: NEAT's Mutation Process. Add connections mutation inserts a new connection node to two existing unconnected nodes. This node, which in this case is node 7, is assigned a random weight value. Add node mutation inserts a new node to an existing connection. In this diagram, node 5 is disabled. Nodes 8 and 9 are added, with 9 receiving the original weight and 8 receiving a weight value of 1 [16].

2.2.2.4 Speciation

Crowding, a phenomenon in which a highly fit genome and its offspring begin to take over the population, often occurs in genetic algorithms [12]. In NEAT, crowding is associated with a loss of innovations, a term which refers to structures or attributes added to the population through the mutation operation [16]. For such modifications to be innovations, it should not exist prior to the current generation. To recall, mutations is how genetic algorithms introduce variation and diversity into the population. However, introducing new mutations to a network initially decreases its fitness, as this destroys desirable structures that were inherited from previous generations. Because of this loss of fitness, innovations will often not survive long enough in the population to be optimized due to new genes being discarded as soon as they are introduced. Not being able to inherit these new genes will lead to less diversity [14].

To maintain innovations and avoid crowding, NEAT categorizes similar network structures into different niches called species. The premise behind speciating the population is to force genomes into competing with other structurally similar individuals in their species. This would allow innovations to remain and optimize through niches in the population, rather than individual genomes [15]. Additionally, further preventative measures are employed to keep one species from taking over the population. To do this, NEAT uses two concepts: explicit fitness sharing and historical markings.

Historical markings is one of the concepts unique to NEAT-derived algorithms [16]. It is the system used to track matching topology among genes during the crossover processes. The idea behind historical markings is that structurally identical genes are derived from the same ancestral origin. For every new gene that is introduced as the result of a mutation operation, the global innovation number is incremented. This value is then assigned to the new gene as its innovation number, which does not change should it be inherited. As historical markings keep track of innovations and matching genes, it can be used to determine topological similarity, which is necessary for speciation. For example, a smaller number of excess or disjoint genes would imply more matching genes and thus shared ancestry and greater similarity. By taking advantage of this idea and specifying some form of hard threshold for similarity, genomes can now be categorized into different niches in the population. In NEAT, the equation used to measure the compatibility distance δ is expressed through the following mathematical formula [14]:

$$\delta = c_1 \frac{E}{N} + c_2 \frac{D}{N} + c_3 \overline{W} \tag{2.4}$$

The compatibility distance value between two genomes is obtained from a linear combination of the number of excess and disjoint genes as well as the average weight difference of matching genes, both enabled and disabled. This is represented

by the variables E, D, and $\overline{W}$ respectively. The coefficients c_1, c_2, and c_3 determine the importance of each of these factors. N is the number of genes in the larger genome of the pair [16].

NEAT keeps an ordered list of species in the population, which allows genomes to be speciated in a sequential manner [14]. Speciation is done on a per generation basis [14]. To determine a genome's compatibility with one of the species in the population, a representative genome of such species is randomly selected from the previous generation. Using a compatibility threshold δ_t, if the resulting δ value does not exceed this threshold, the current genome is therefore compatible with the representative genome and will be added into the species. However, if the resulting δ value crosses the threshold, a new species will be created with the current genome as its only member. In the case that a genome is compatible with multiple species, it is placed in the first compatible species in the ordered list. The steps involved in speciation are summarized in the Genome Loop pseudocode below [16]:

Algorithm 2 The Genome Loop

for genome g in population P **do**
  for species s in list S (The Species Loop) **do**
    if $\delta > \delta_t$ for all species in S, generate new species s_{new} with g as its member **then**
    else
      if $\delta < \delta_t$ for s, place g in s **then**

Another prevention measure for crowding and loss of innovation is explicit fitness sharing, which manages the size of each species. Explicit fitness sharing is implemented to ensure that species with high performing genomes are not able to dominate and take over the population. In explicit fitness sharing, genomes in the same species share fitness. This means that species have to avoid becoming too large as the average fitness will decrease as a result of having to share among

too many genomes. The mathematical formula for adjusting the fitness value of a genome is given as follows [16]:

$$f_i' = \frac{f_i}{\sum_{j=1}^{n} sh(\delta(i,j))} \tag{2.5}$$

f_i represents the original fitness of the genome i, whereas f_i' represents the adjusted value. The sharing function sh returns 0 or 1 depending on whether the compatibility distance between i and some genome j in the population is above or below the threshold δ_t (respectively), which means that $\sum_{j=1}^{n} sh(\delta(i,j))$ represents the number of genomes belonging to the same species as i. Thus, the adjusted fitness value is simply the original fitness divided by the number of genomes in i's species [14]. Given the adjusted fitness values for each genomes, the new size of a species is recalculated every generation using the following mathematical formula [15]:

$$N_j' = \frac{\sum_{i=1}^{N_j} f_{ij}}{\overline{f}}, \tag{2.6}$$

where N_j' and N_j denote the new and old number of genomes in species j respectively. The new size N_j' is calculated by dividing the sum of adjusted fitness values f_{ij} of genomes i in the species j by the mean adjusted fitness of the entire population $\overline{f}$. Upon finding the new size N_j', the algorithm then randomly mates the best-performing r percent of the species. This process generates N_i' offspring, which are then used to replace the species' entire population and thus eliminate low-performing individuals. Through this formula, the effects of explicit fitness sharing can be seen: species grow when their average adjusted fitness is above the population's average, or shrink when the reverse is true [15].

2.2.2.5 Real-time Implementation

NEAT is highly effective against other neuroevolution methods when it comes to complex tasks such as the benchmark double pole balancing task [15]. Thus, it can be adapted to work in real time for video games. As NEAT is an offline algorithm, some modifications are required for it to function in an online setting. In scenarios such as the game NERO, game agents are treated by rtNEAT as genomes in the population [16]. In order to behave dynamically, genomes must continuously evolve and be replaced at the same time as fitness evaluations [16]. This is not the case for the original NEAT algorithm, which only replaces the current generation of genomes once it is done evaluating the entire population [16]. Depending on the type of game and the player's expectations, this strategy may still work. However with games like NERO whose seamless evolutionary mechanic plays a central part, replacing the entire population all at once may feel unnatural to the player and ruins immersion as the changes are too obvious.

rtNEAT mitigates this by implementing an additional feature that constantly replaces an individual in the current population after a certain amount of time. Usually this time interval is relatively short so that the replacement feels invisible to the player. Furthermore, as only a single individual with poor fitness is replaced at a time, the changes feel more gradual and natural to the player. The algorithm for this new feature is demonstrated in Algorithm 3 below [16]:

Algorithm 3 rtNEAT Replacement Cycle

for every n ticks of the game clock **do**
    Calculate adjusted fitness value for all genomes in population
    Remove genome with worst adjusted fitness after a sufficient amount of time has passed
    Re-calculate average adjusted fitness for each species
    Choose two genomes from a parent species and create offspring
    Adjust dynamic compatibility threshold, reassign genomes to species
    Replace removed genome with new offspring

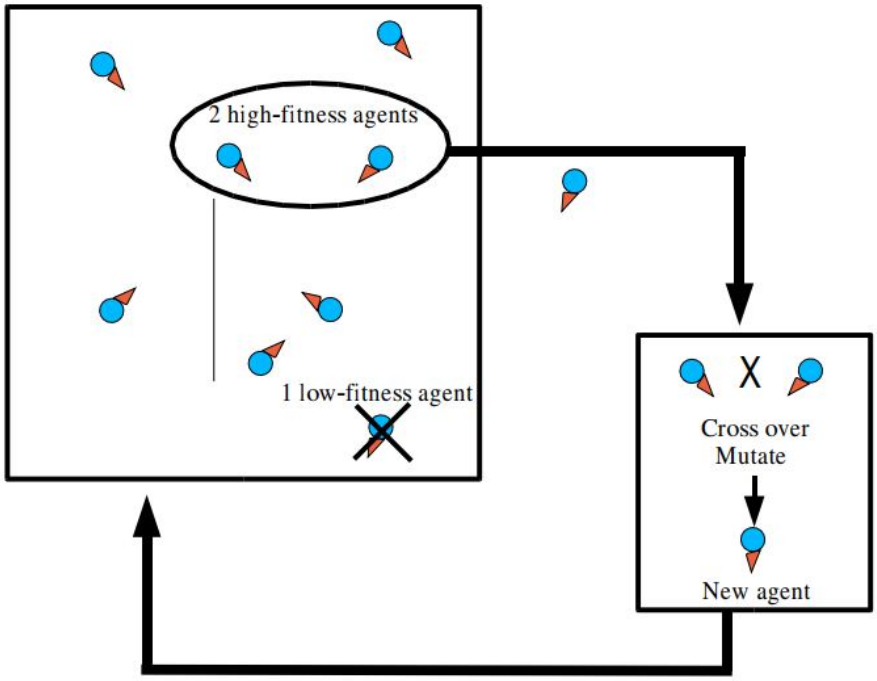

Figure 2.7: rtNEAT's continuous removal and replacement cycle [16]

rtNEAT, despite being modified with many changes, still retains the speciation, complexification, and protection of innovation that was key to the original algorithm. It is important to specify that only the genome with the lowest adjusted fitness is removed from the population. This prevents new topological mutations from being removed prematurely. Removing genomes with lowest unadjusted fitness does not take fitness sharing or speciation into consideration, and thus innovation will no longer be protected. Additionally, time is also taken into account when deciding the worst genome to be removed. In a real-time context, each agent is introduced into the population at different time periods. In order to protect innovation, it is important for such agents to remain for a sufficient amount of time before removal. This will allow new genes to persist among the population through mating and inheritance [16].

Offspring creation in rtNEAT also retains the speciation dynamics of the original algorithm. Parent genomes are randomly chosen from high performing species, which are chosen probabilistically based on how their average adjusted fitness values compare to the overall population average. As for species reassignment, rtNEAT uses a dynamic compatibility threshold that slightly raises or lowers depending on whether there are too many or too few species in the population. This keeps the number of species in the population consistently stable. In addition, rtNEAT does not reassign genomes every time a replacement occurs. In NERO, for example, species are reorganized after every five replacements [16].

A unique problem that arises with implementing rtNEAT is determining the time interval, or n ticks, between each replacement. Using a law of eligibility, an appropriate value can be determined. This is expressed by the following mathematical formula [16]:

$$I = \frac{m}{|P|n} \tag{2.7}$$

where I represents a fraction of ineligible agents in the population, m denotes the minimum time before eligibility, n is the time between each replacement, and $|P|$ is the size of the population. To recall, rtNEAT takes into consideration the age of each genome in the population. Given a minimum time m, agents are only eligible for replacement when its age exceeds this threshold. According to this law of eligibility, the fraction of ineligible agents I lower with a smaller m threshold or a larger population and n value. Whether or not I should remain small depends on various factors; the mating pool minimizes when I is larger, but too many eligible agents can affect the game's performance due to limited CPU resources [16].

2.2.3 CIGAR

Case-Injected Genetic Algorithm, also known as CIGAR, combines genetic algorithms with long term case-based memory for better performance and faster problem-solving. The idea behind CIGAR is to allow genetic algorithms to gain experience and apply previous knowledge to solve a similar problem rather than starting from scratch. To recall, genetic algorithms typically begin with a randomized population of hypotheses to ensure diversity and avoid search bias. Such bias, however, may be beneficial for systems that are frequently required to solve similar or related problems. One of such systems is video games, the domain of our research. From a simple game like Pac-Man to an open world like that of Skyrim, game mechanics often revolve around a repeated set of actions. Such actions may be slightly altered or expanded upon for variety and difficulty adjustment, but they rarely change completely throughout the course of the game. Additionally, computational performance is a big area of interest in the domain of video games. Resource limitations often play a big part in not only determining accessibility and player enjoyment but also game design. As implementing machine learning methods such as genetic algorithms may take up significant resources, CIGAR's case-based memory can greatly improve game performance through faster problem-solving. Furthermore, CIGAR's faster performance is necessary for genetic algorithms to function as a game mechanic or work concurrently with gameplay, such as in the game NERO from the rtNEAT study.

Overall, CIGAR has been shown to not only learn from previous experience but also produce faster and better solutions compared to traditional genetic algorithm techniques [10]. These results were obtained over a series of 50 problems. In the study conducted by Louis and McDonnell, sets of 50 similar problems were generated based on three different domains: combinational circuit design, strike force asset allocation, and job shop scheduling. CIGAR was applied to such problems

with the goal of demonstrating improved quality of solutions and performance in mind. In the study, quality of solutions is determined by the best fitness values obtained from solving each of the 50 problems. Meanwhile, performance is dictated by the time taken to solve each problem as well as number of generations taken to find the best solution [10].

2.2.3.1 Algorithm

As illustrated in both Fig. 2.8 and Fig. 2.10, the CIGAR system can be seen as being composed of interactions between the genetic algorithm and case-based reasoning modules. CIGAR works by periodically introducing appropriate solutions, both full and partial, from previously solved problems into the population to quickly find solutions without having to start from scratch. Such solutions are taken from the case base - a database that stores and supplies the genetic algorithm with problems and solutions, the latter of which is what makes up the term "cases". To be specific, a case in CIGAR refers to a candidate solution genome with additional information on its fitness value as well as when it was created. CIGAR doesn't need to begin with a predefined set of cases; the case base is able to boot itself and grow over time through the genetic algorithm module's problem solving attempts [10].

There are two ways of approaching the CIGAR algorithm. The first approach, referred to as CIGAR$_p$ in the Louis and McDonnell study, injects cases based on a problem similarity metric. This approach is based on the assumption that similar problems will have similar solutions. As illustrated in Fig. 2.8, this version of CIGAR works by first finding similar problems in the case base then injecting a few of their solutions into the genetic algorithm's initial population. In addition to the small group of injected solutions, the rest of the genomes in the population are randomly initialized to ensure diversity. Afterwards, the genetic algorithm carries out its search as usual, storing any good partial solutions it comes across into the

case base through a preprocessor. Note that these solutions are saved from different generations and vary in fitness. Such solutions consist of genomes with the highest fitness in their current populations. This process works cyclically every time the system is given a new but similar problem to solve [10].

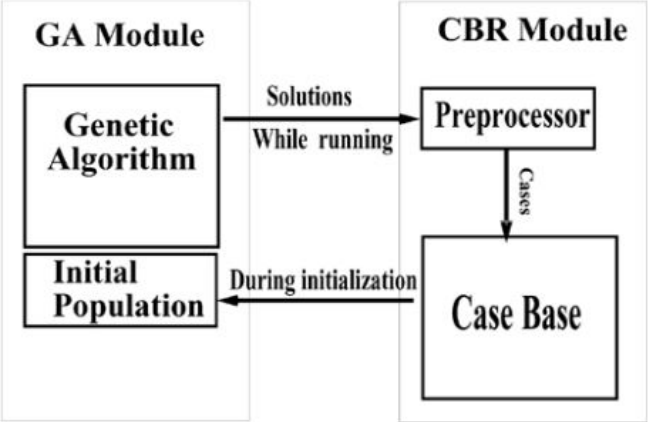

Figure 2.8: Conceptual diagram of CIGAR$_p$ [10]

To examine how CIGAR$_p$ works in the long run, let there be n problems that need to be solved sequentially, as illustrated in Fig. 2.9. Each problem is denoted by P_n. Prior to solving the problems, the case base is initially empty. Starting with P_0, a genetic algorithm is initialized with a random population of genomes. As the algorithm attempts to solve the problem, the system generates cases to be stored in the case base. At this point, it will not attempt to inject solutions from cases into the population. Moving onto P_1, it will once again generate new cases to be added to the case base. However, this time it will also attempt to inject cases from P_0. For problem P_2, it should be noted that while the process is repeated, the case base now has cases from both P_0 and P_1 for the system to choose from. Thus, it can be generalized that for an arbitrary problem P_n, the system injects cases from P_0 to P_{n-1} into its population [10].

When applied to a parity checker design problem, Louis and McDonnell found certain patterns that arose in CIGAR$_p$'s behavior. Firstly, for any two problems that

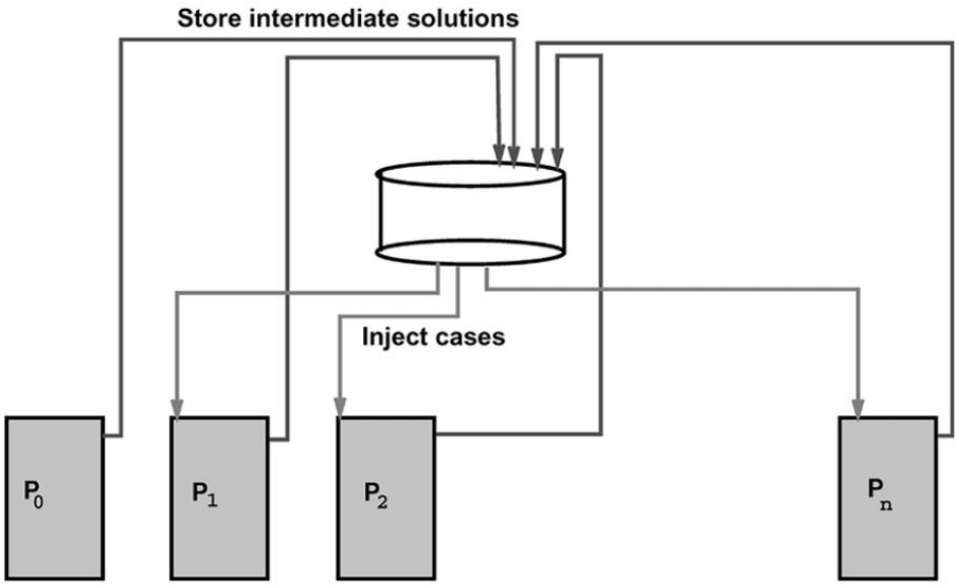

Figure 2.9: Solving n problems with CIGAR$_p$ [10]

are far apart, CIGAR$_p$ performs better when it injects into P_{new} less fit solutions from P_{old}. This pattern is further exaggerated in larger sized problems. Secondly, injecting cases with higher fitness in P_old tend to lead to a quicker flattening of the learning curve over time [10].

The second approach to CIGAR - CIGAR$_s$ - works under the assumption that similar solutions will have similar problems. For each problem, CIGAR$_s$ periodically removes genomes with lowest fitness and replaces them with solutions from the case base that are similar to the current best genome. Unlike CIGAR$_p$, injection and generation of cases occurs repeatedly within a single problem, as illustrated in Fig. 2.10. The algorithm for CIGAR$_s$ is given below [10]:

This pseudocode adapts the standard steps involved in a canonical genetic algorithm but performs additional injection and caching steps. These steps are carried out after a certain time interval and not during every iteration of the while loop. The period of time stored in the variable *injectPeriod* is determined by the experimenter. It is important to choose a sufficiently large time interval for the

Algorithm 4 CIGAR$_s$ algorithm

$t = 0$
Initialize $P(t)$, which is currently at the first generation.
while termination condition has not occurred **do**
  if $t\%injectPeriod == 0$ **then**
    Perform $InjectFromCaseBase(P(t), CaseBase)$
  end if
  Generate the next generation's population $P(t+1)$ using selection, mutation, and crossover techniques
  $t = t + 1$
  if $NewBest(P(t))$ **then**
    Cache the new best individual
  end if
end while
Save cached genome into case base
  =0

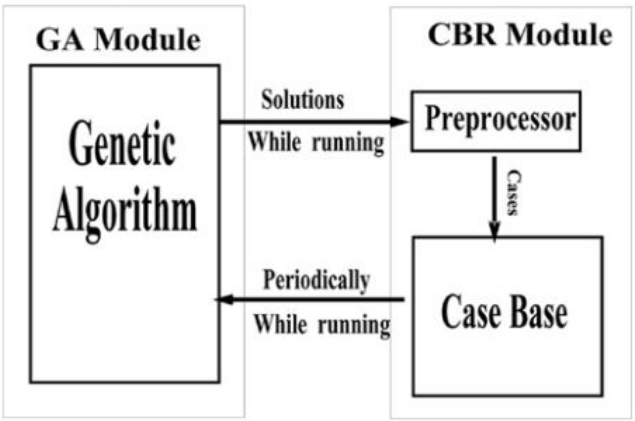

Figure 2.10: Conceptual diagram of CIGAR$_s$ [10]

genetic algorithm to make some progress before case injection and population replacement occurs. After injecting cases and removing a small, low performing part of the population in function *InjectFromCaseBase*, the genetic algorithm goes on to create the next generation using the standard selection, crossover, and mutation techniques. At this point, should the injected individuals lead to a new best genome or an increase in the maximum fitness value of the population, such genome will be cached then later saved into the case base [10].

2.2.3.2 Determining Similarity

For CIGAR to work, it is important for the injected solutions to be relevant - this means simply injecting random solutions will not provide good results [10]. As two different versions of CIGAR exists, there are two different ways of determining similarity.

The first approach is based on problem similarity. To recall, CIGAR$_p$ works under the assumption that similar problems have similar solutions. The algorithm focuses on finding cases from similar or relevant problems in the case base before injecting their solutions into the population. For this approach, determining a problem similarity metric is not only necessary but also non-trivial, according to previous research [10]. However, defining such a metric is often difficult. Problem similarity depends on the problem or domain at hand, and each domain will have a different similarity metric [10]. Furthermore, genetic algorithms are often applied to poorly understood domains or problems where predetermined sets of inputs and outputs are not necessarily available [10].

While having a similarity metric is necessary when using this approach, making it exact is not. An inexact, problem-independent similarity metric may still work as long as a wide variety of solutions are stored in the case base or injected into the population [10]. This means solutions with varying levels of fitness taken from different generations. By effectively covering all bases, CIGAR$_p$ lets the genetic algorithm module determine relevant solutions by assigning such solutions fitness values. This is the more realistic and widely applicable approach [10]. Should none of the injected solutions be useful, they will have low fitness and be eliminated from the population over time. Thus, the system will fall back into a purely genetic algorithm based approach. With the case-based reasoning module gone, CIGAR's performance will definitely decrease, but it will generate a solution nonetheless [10].

The second approach is based on solution similarity. Typically, solution similarity

in CIGAR determines what cases to be injected based on their similarity or closeness to the current best genome in the population [10]. This is often much simpler than determining problem similarity, as the system can take advantage of the genetic representation used by the genetic algorithm [10]. Genetic algorithms commonly uses string or binary representations, and determining similarity between strings is relatively easy [10]. This similarity is also domain independent, unlike the problem similarity metric of the first CIGAR approach [10]. This makes CIGAR$_s$ the preferable approach, and research has shown good results from having a similarity measure that is solely syntactic [10]. However, this does not mean that the similarity metric of CIGAR$_s$ is nontrivial. Similar to the CIGAR$_p$, solution similarity metrics can be inexact and noisy [10]. However, due to the way genetic algorithms function, unsuitable cases will be quickly eliminated from having low fitness. In addition, injecting cases with an inexact metric can still aid with maintaining diversity in the population [10].

In the study conducted by Louis and McDonnell on sets of 50 problems, injection of relevant solutions was determined probabilistically based on syntactic solution similarity. The following mathematical formula was used [10]:

$$Prob(C) = \frac{l - D(I, C)}{\sum_{i=1}^{i=n}(l - D(I, J)} \tag{2.8}$$

The formula describes the probability of injection as being proportional to its similarity to the current best member of the population. I denotes the best performing genome, whereas C represents the case being considered at hand. Additionally, l stands for the length of the bit string representation and n represents the amount of cases in the case base. D in this case is the function used to determine the number of bits that differ between two binary strings, otherwise known as the hamming distance. This function is given by the following formula [10]:

$$D(A, B) = \sum_{i=0}^{i=l} A_i \oplus B_i \tag{2.9}$$

where l remains the genome length and $\oplus$ represents the exclusive or operator. As the hamming distance determines the number of bit positions at which the values of two bit strings differ, $l - D$, in reverse, is used to find syntactic similarity between two genomes [10].

Using an inexact or problem-independent similarity metric, different strategies can be applied depending on the closeness of solutions from two given problems. For example, if solutions differ more, cases from earlier generations are more useful. Conversely, if their solutions differ less, cases from later generations should be used instead.

2.2.4 ACTB

Advocates and Critics for Tactical Behaviors (ACTB) is a genetic algorithm originally created for controlling unmanned ground vehicles, but was later adapted to address the need for flexible and believable NPC behavior in video games [6]. Like rtNEAT, it adapts genetic or evolutionary techniques to real-time problem-solving. This is necessary as games are becoming more open-ended, and what makes them enjoyable no longer relies on limited sets of objectives, rules, or roles [6]. Instead, a large number of games now depend on the simulation and immersion seen in first person shooters or open-world games. With a more open-ended or simulative approach to game design, there is a growing expectation for NPCs to produce not only useful but also believable behavior, which was not the case in the past. Furthermore, having dynamic NPC behavior is necessary to avoid predictability, which was often an issue in older games that rely on scripted behaviors.

ACTB at a higher level functions as a control interface or a genetic computational

human behavior model (HBM). Attempts at applying static, non-evolutionary HBMs to control NPC behavior has been previously explored with mixed results [6]. A HBM-based NPC controller decides, in real-time, what actions an NPC should perform based on a combination of internal metrics and factors available in game [6], such as game rules or points. While achieving complex NPC behavior is possible for static HBMs, maintaining such level of complexity while ensuring effectiveness is not an easy task without machine learning [6]. Furthermore, the core aspects of good and effective NPC behavior should not be sacrificed for excessive complexity or poor performance. In addition, complexity is not the only factor when it comes to creating dynamic and believable behavior, as addressed in the following section. ACTB alleviates these issues by including an evolutionary component to the HBM. ACTB works by controlling NPCs with continuous re-planning cycles in addition to internal game logic, allowing them to adapt to changing conditions and strategize against human players as the game goes on. Each genome in the population represents a plan that can be changed through traditional crossover and mutation operators as well as the additional flagging system explained in the following sections. Plans are evaluated against a set of criteria that determines their fitness and thus effectiveness [6].

2.2.4.1 Metrics

Good NPC behavior at its core is comprised of three different criteria: the ability to achieve objectives, to perform in an efficient manner, and to be interesting. The first criterion, which is being able to achieve objectives or to "play well", is what makes an NPC useful to a game. In many games, this is easily achieved through strict or straightforward rules and winning conditions, such as that of backgammon, which simplifies the roles NPCs play. The second criterion, which is to perform efficiently and intelligently, requires the game to have NPCs not only achieve their objectives

but also do so in the right way. For example, an enemy NPC should not have to do any excessive or unnecessary actions before shooting at the player. Determining effectiveness depends on the mechanics of each game, but may include factors such as resources consumed or the amount of damage taken. The third criterion, which is to perform tasks in an interesting way, is much more subjective than the other two as it heavily depends on the player's perception. This subjective quality can be determined through numerous factors including unpredictability, interpretability, or variety. In older video games, the first two criteria are much more applicable and perhaps easier to achieve due to their predictable and scripted gameplay. The third is more achievable for games with a simulative and open-ended approach [1].

One of the factors that can be used to determine interesting behavior is believability. This factor is the key idea behind the development of ACTB. What makes a behavior believable is how human-like it is. In order to create believable behavior, human behavior has to be broken down and understood to be emulated. Hussain and Vidaver define believable behavior as being comprised of three broad categories - plan, act, and react [6]. Such categories are used by human players to judge NPC behavior. The plan category includes being able to strategize, coordinate actions with other players or NPCs, and recognizing failed attempts. The act category expects abilities and timing that are human-like or within human capabilities. The last category, react, includes reacting to different factors and agents in the game appropriately. This includes players, changes in environment, foes, and allies [6].

Believability forms the basis for much of the features and inner workings of ACTB. The categories of believable behavior act as principles that guide the set of criteria or objectives plans are evaluated against. Such objectives form a unique feature called critics, which comprises of [6]:

- ObjectiveSuccess, which determines whether the planned path successfully reaches the objectives of the game.

- Safety, which deals with the amount of risk involved in the plan. Risk is often identified as being within the enemy's line of sight.
- Traversability, which relates to the amount of obstacles present in the path.
- Duration, which pertains to the time it takes to completely traverse through a path.
- TimeToObjective, which is the time to complete a certain objective.
- ComplexPath, which deals with the complexity of a path, i.e: the number of segments or branches present in it.
- Skulk, which measures visual exposure. This is different from Safety, as an NPC can be visually exposed but still safe from the enemy.

It should be noted that each critic as defined in the original research conducted by Hussain and Vidaver is meant for games with NPC path planning and stealth or aggression mechanics [6]. However, such critics can be easily altered to suit the needs of other game types. Critics can also be added or removed. To ensure that NPCs can react in real-time, however, there is a limit to the number of critics added to the ACTB controller [6].

2.2.4.2 Framework

Fig. 2.11 summarizes the workings of ACTB at a higher level. ACTB depends on a real-time cycle of replanning that constantly changes and introduces new plans into the system. The two key components involved in this process are critics, which are criteria that determines a plan's effectiveness (as explained in section 2.2.4.1), as well as advocates, which make changes to previous plans and creates new plans according to the system's current knowledge of the game world. In ACTB, plans consist of sequences of waypoints or paths leading up to an in-game destination as

well as actions to perform at every step along the way. These actions may consist of avoiding enemies or obstacles, collecting key items, and so on. In each cycle, ACTB re-ranks plans according to their effectiveness using the critics model, and chooses the best plan to be executed in game. A portion of the remaining plans then undergo changes via the advocates component, and the cycle restarts once more [6].

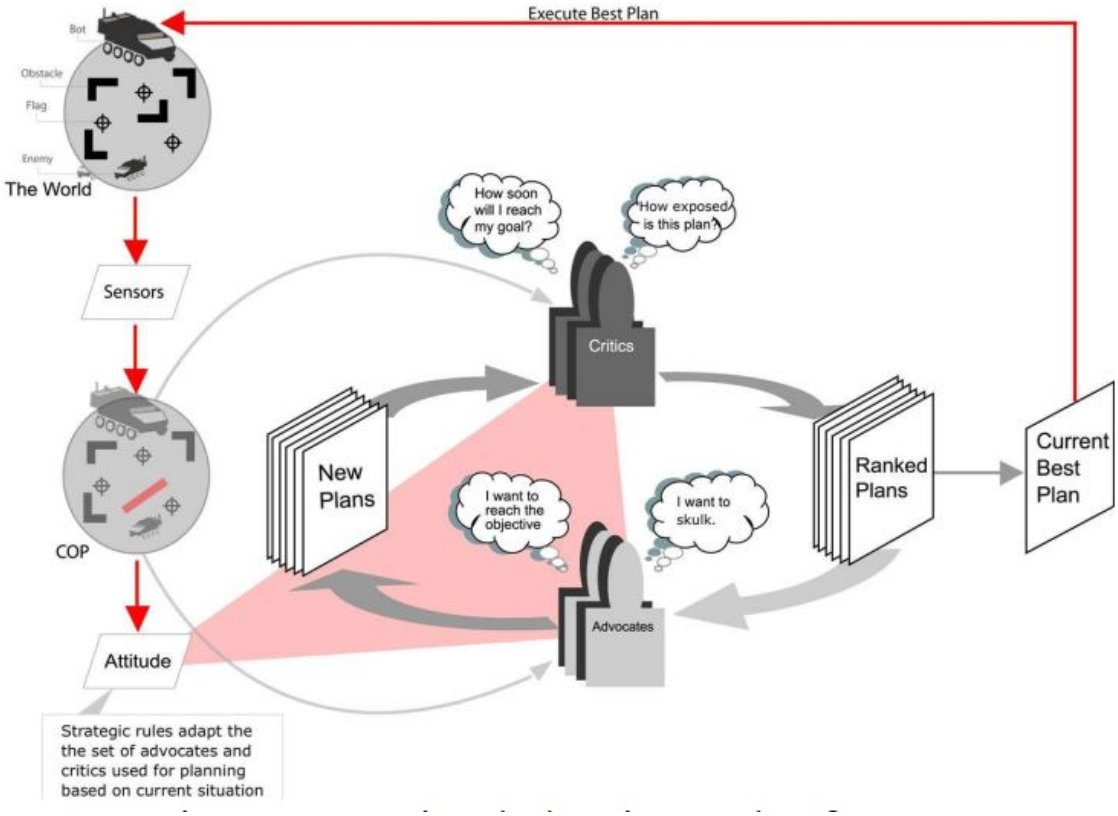

Figure 2.11: Overview of the ACTB cycle [6].

Like all genetic algorithms, ACTB performs a search through the space of possible genomes or solutions. For ACTB, such genomes are plans for NPC movements or paths. Such plans are evaluated against critics, whose weighted sum make up their fitness values. Creating new genomes or plans is mainly managed by the advocates component. Alongside the canonical crossover and mutation techniques, the advocates component introduces new plans into the population via a flagging

system. This system involves evaluating plans against critics criteria and marking problematic parts of the genome that need change [6].

Detecting changes to the game world and updating the system's knowledge is done cyclically. With large quantities of new information constantly available in game, it can be difficult for the system to keep track of changes and form plans that respond to them in a timely manner. Furthermore, taking into consideration that each cycle of the system involves updating the entire population, having too many factors to consider at once would reduce performance significantly. In order to effectively handle such changes in real-time, ACTB utilizes categories of reactions or behavioral themes. Such themes, termed as attitudes, include Brave, Scared, Cautious, and Ambitious. They are illustrated in Fig. 2.12 below. In response to changes to world conditions, ACTB identifies the appropriate attitudes used by critics and advocates to plan accordingly. Each attitude prioritizes and rewards different critics, for instance a Scared attitude would give higher weight to the Skulk or Safety critic [6].

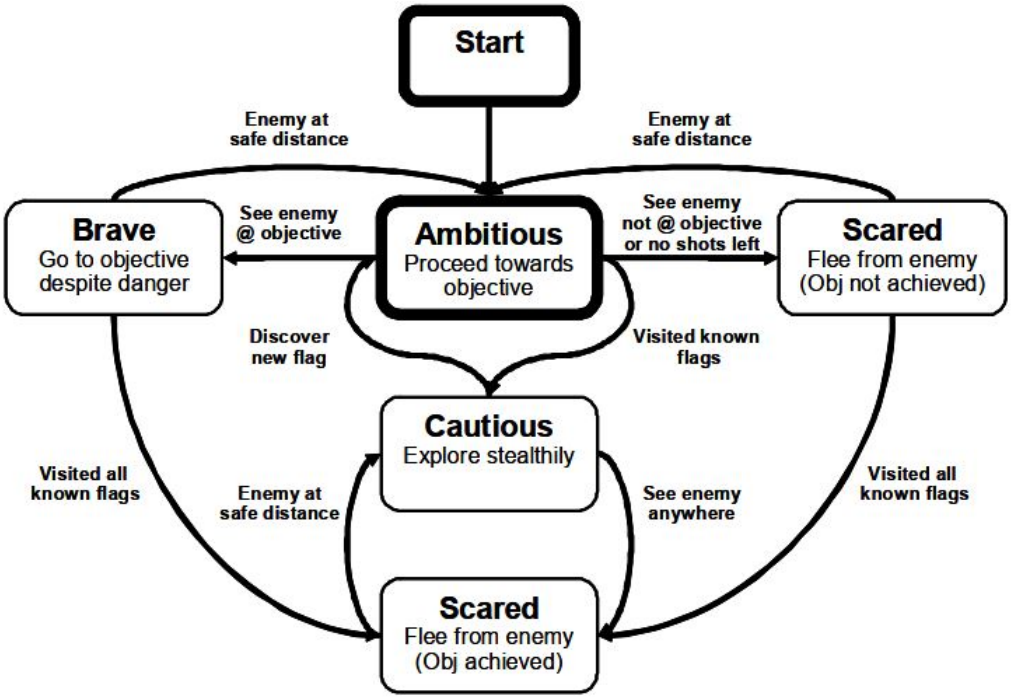

Figure 2.12: Decision making through attitudes in ACTB [6]

CHAPTER 3

Software

The aim of the software portion of this Independent Study is to demonstrate how genetic algorithms can determine the right kinds of NPC behaviors that would suit the player's skill level. Thus, a game with NPCs is implemented along with a genetic algorithm that is developed specifically for the game's mechanics. This chapter is divided into two main sections to discuss features of the base game and its genetic component.

3.1 Game Overview

The game developed for this Independent Study is inspired by Pac-Man, an influential maze chase video game. Pac-Man was chosen as the main source of inspiration due to its 2D graphics, straightfoward core mechanics and high replay value. Additionally, this game has less graphical and computational requirements allowing for it to perform well on most machines and lessen the performance impact that may stem from its genetic component, despite using a genetic algorithm with binary representation is typically fast. Furthermore, Pac-Man's gameplay mechanics are simple yet sufficiently complex to give rise to many different possibilities for genome representations. Given the straightforward and well-established nature of Pac-Man NPC's, any changes to their behavior can be easily identified. Having too many or too complex mechanics may make changes in difficulty hard to quantify. In

terms of replay value, the games' replayability provides an infinite ceiling for growth in both challenge and skill, which is necessary to maintain a state of flow according to Cruz's and Uresti's model [2]. Additionally, if behavior of NPCs change slowly and require many iterations of the genetic loop, the game's replayability can help mask these issues and allow the algorithm ample time for correction. Furthermore, it is a game that contains enemy characters with a variety of different behaviors and states that could be useful for an attitude-based approach to genetic algorithms, similar to that of ACTB. The amount of enemy characters in the game may also work in a real-time approach similar to that of rtNEAT where each character acts as a separate genome in the population, although typically in genetic algorithms ideal population sizes are much bigger [13].

For the purpose of this Independent Study, some changes have been made to the gameplay mechanics of the original Pac-Man:

- Firstly, the game does not implement "modes" for the enemy characters. In the original Pac-Man, all four ghost characters behave according to three different modes - Chase, Scatter, and Frightened. To summarize, ghosts often spend most of their time alternating between Chase mode, which involves following Pac-Man based on the player's current tile position, and Scatter mode, which leads them to disperse. Frightened mode is uniquely triggered by the player character consuming one of the "Power Pellets" or "Energizers" which are represented by larger dots on the map. In Frightened mode, ghosts move slower and randomly, allowing them to be eaten by Pac-Man.

  In our game, ghost characters are always in Chase mode. This decision was made to simplify the game, provide a testbed that displays clear qualitative results, as well as accommodate the workings of the genetic algorithm. Thus, without the genetic component, it is important to maintain balance and prevent ghosts from overpowering the player. With four ghosts constantly being in

Chase mode, they can easily catch up to and corner the player. Their default speed is therefore set to a value of 3 compared of that of 8 from the player. In our game, a lower number represents slower speed, whereas a higher number represents faster speed.

- Power Pellets in Pac-Man forces all ghosts on screen to go into Frightened mode. This mode, along with Scatter mode, however, are both removed to simplify the game for testing and implementation of the genetic component. With only Chase mode implemented, the game's Power Pellet feature is therefore removed. Collecting small pellets will generate 1 point, while large pellets will give the player 10 points.

- Similarly, there will be no "Bonus Fruits" feature for simplicity. In the original Pac-Man game, Bonus Fruits refer to collectibles that give players additional points in addition to the pellets available on screen. They appear near the starting point of the ghosts' paths and tend to disappear after a set amount of time. This enables a "High Score" feature, which does not exist in the game developed for this Independent Study.

- The score system of the game keeps track of how far the player is from collecting all of the available points on the map. This provides a clear winning condition for the player to achieve which matches one of the conditions for inducing flow - having unambiguous demands for action and feedback. Furthermore, having a capped maximum score can inform the genetic algorithm the player's progress towards victory, which could be useful information for its fitness function. Additionally, there is now more emphasis on the performance and behavior of the enemy ghost characters in determining the game's replay value.

- The health system does not exist in the game implemented for this Independent

Study. The original Pac-Man gives players "extra lives" - several chances for the player to keep playing after colliding with the enemy. No penalties are applied and all progress (e.g, pellets consumed, points, time recorded, etc.) is saved upon restarting the game through an extra life. The game only reaches the game over state when all lives are lost. To simplify the game's mechanics and shorten single playthroughs, our game implements an "instant death" feature. This means that no progress is saved upon player character death, and the game instantly triggers a game over state. The reason behind the shortening of playthroughs is to accomodate the workings of the genetic algorithm, which performs fitness updates at the end of each run during win or game over states. An in-depth explanation on the genetic algorithm will be explained in the following sections.

- A timer is added to our game alongside a score tracker to implement the fitness function. In the original Pac-Man game, a timer does not exist. The timer serves to inform the algorithm of how fast the player character is killed by a ghost, whereas the score tracker keeps track of how far the player is from victory. Both of these factors can be taken into consideration by the genetic algorithm's fitness function.

- In the map of our game, no "warp tunnels" exist. In the original Pac-Man, this feature denotes two special tiles located in the middle portion of the map. These tiles are aligned and placed in two opposite sides. They appear open instead of closed like the rest of the tiles that surround the edges of the maze. These tiles serve to allow Pac-Man to travel to the opposite side of the map, which is a good mechanic to have to balance the difficulty of the original game. In order to compensate for the lack of warp tunnels and Frightened or Scatter modes, the speed of the ghosts have been slowed down significantly in the

static version of our game. This is to create more distance between the player and enemy characters as well as allow players more time to strategize and plan their next move.

- The original Pac-Man has a total of 256 levels and features the same map throughout the game. For this Independent Study, only one level and one map is created as a control variable. This helps to accurately compare the differences in difficulty brought by changing ghost behaviors and speed in each playthrough.

Both parts of the software - the game and the genetic algorithm - are developed in Unity using C#. The original game without the genetic component is stored in the "Game" scene of the project. Meanwhile, the version that incorporates the genetic algorithm can be accessed through the scene "Game GA". In order for the restart button to function, simply comment out the respective code block for one of the scenes in function "RestartButton" of HUDController.cs.

The following subsections explain the components that make up the game's core mechanics. Although referred to as "controllers" or "managers" at a higher level, each component is implemented as a separate class in code form.

3.1.1 Node Controller

Nodes, which are single squares of a grid, form the underlying organizational structure of the game. This is to mimic the way the original Pac-Man was implemented. This grid system allows both player and enemy characters to travel around the maze easily on a node to node basis. Additionally, enemy characters can track down and follow Pac-Man easily without the need for raycasting or line of sight techniques. Furthermore, in terms of designing the maze, the grid system makes it

easy to standardize and restrict the height or width of paths to the dimensions of a one-by-one square.

The Node Controller class is attached to each node on the map, which altogether forms a maze. For each node, the controller keeps track of all four neighboring nodes (based on cardinal directions) and whether they are valid positions for a character to move to - for example, whether a neighboring node is a wall or a part of the path, marked by "false" or "true" respectively. In addition to keeping track of neighboring nodes, the Node Controller also checks whether its node contains a pellet or an energizer. Most nodes contain a pellet by default, but special nodes such as the Ghost Starting Nodes (explained in section 3.1.1.1) of the static game do not.

In order to keep track of neighboring nodes, two variables - one boolean and one GameObject variable - are provided for each cardinal direction (left, right, up, down). For clarification, a GameObject is a class specific to Unity, and holds reference to an instance of an in game object - in this case, that is a nearby node object. The process of checking for neighboring nodes first involve raycasting, which is performed for all directions in the start of the game. A boolean variable for a certain direction is set to true if a valid neighboring node exists for that direction. GameObject variables will then store reference to the node object so that it can be accessed by the method GetNextNode as well as the Movement Controller. The method GetNextNode receives a direction and returns the node that exists in such direction.

By default and at the start of the game, most nodes have a pellet boolean variable set to true and an energizer boolean variable set to false. If a node has the pellet variable set to true, it contains a pellet. The Node Controller will then check for an "Energizer" tag attached to the pellet object. If there is an "Energizer" tag set for the pellet object, the pellet is an energizer. Thus, the energizer variable is also set to true.

3.1.1.1 Ghost Starting Nodes

Ghost Starting Nodes are special nodes that provide starting positions for the ghost characters in game. This mechanic is directly taken from the original Pac-Man game. In the static game, housing the ghosts in these nodes gives the player some time to move away from the enemy characters at the start and thus balances the game's difficulty. There are four different starting nodes denoted by the names "Left", "Right", "Center", and "Start". "Left" refers to the leftmost node within the box that houses the ghosts at the start of the game. Similarly, "Right" refers to the rightmost node within the box. "Center" refers to the node between "Left" and "Right". "Start" is located above "Center" and within the paths of the maze, in between two of the normal nodes containing pellets. To leave this box, the ghosts follow the order of "Left" or "Right" node to "Center" node and finally "Start" node, as illustrated in Fig. 3.1 . For example, ghosts positioned on the "Left" and "Right" starting nodes move towards the "Center" node and upwards to the "Start" node. Meanwhile, the ghost on the "Center" node simply has to move to the "Start" node. From the "Start" node, they would then move along the paths of the maze to chase the player. This feature is only implemented in the game's static version, as it was later discovered that the different starting positions may impact a ghost's fitness, which also takes into account its average distance away from the player.

3.1.2 Movement Controller

The Movement Controller is built on top of the Node Controller. It is attached to both enemy and player characters on screen. Whereas Node Controller is in charged of organizing and linking nodes in the map, the Movement Controller is the component that directs characters to move in between such nodes. A character's Movement Controller also contains details regarding its movement speed, and in

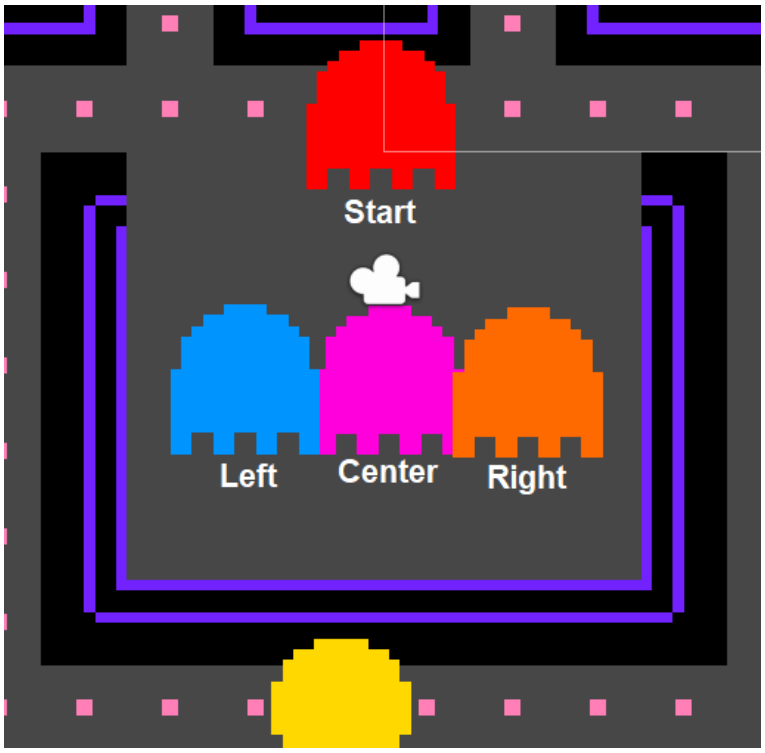

Figure 3.1: Initial ghost movement pattern from the static version of the game.

the case of the player is also able to remove pellet objects once they have been consumed. Using the GetNextNode method provided in the Node Controller, the Movement Controller makes it possible for additional, character-specific controllers to simply specify desired directions in the form of strings (e.g, "right" or "up") for a character to follow. More details regarding these character-specific controllers will be provided in the following sections.

A Movement Controller contains the following variables or objects: the GameObject "nextNode" to store the next node to move to, a floating point number denoting the character's speed, strings for both the current and next directions, as well as a boolean variable isGhost that is set to true for enemy characters. This controller functions through the Update and FixedUpdate functions provided by Unity. Detection of input is done on Update, whereas FixedUpdate handles the transformations to simulate continuous character movement. This is because Update is not guaranteed

to call in fixed intervals while FixedUpdate is. Update is then used as a "steerer" that constantly checks whether a character is directly aligned on top of a node and tells them which node to go towards next depending on whether a node exists in a given direction. FixedUpdate then comes in to physically move the character towards that next node. Note that the given direction does not change until a new direction is specified through an additional character-specific controller. This creates the effect of characters to continue to move in one direction until a new direction is specified, similar to the way characters move in the original Pac-Man game.

3.1.3 Player Controller

The Player Controller is an additional character-specific controller. Its purpose is to receive keyboard inputs and use them to dictate the player character's next direction. In the game, the player can choose to use both "WASD" or arrow keys control schemes to move their character. "W" or the up arrow key is used to direct the character towards the northern cardinal direction. "A" or the left arrow key is used to direct the character towards the west. "S" or the down arrow key is used to direct the character towards the south cardinal direction. Lastly, for "D" or the right arrow key, the player will be directed towards the eastern direction. For each keyboard input, their respective cardinal direction is saved into a string which is then fed to the player character's Movement Controller through the method SetDirection.

3.1.4 Enemy Controller

Similar to the Player Controller above, the Enemy Controller is an additional controller specialized for enemy or ghost character behaviors. Unlike the player character, however, the movement of ghosts are not dictated by keyboard inputs.

Enemy characters move autonomously around the maze using algorithms. Additionally, enemy characters do not consume or interact with pellets like the player character does. Instead, they have a unique feature that allows them to cause a game over event upon colliding with the player character.

Similar to the original Pac-Man, four different types of enemies exist in this game - red, blue, pink, and orange ghosts. Although the goal of all enemy characters is to chase the player character down, they each do so in a slightly different manner. The different algorithms implemented for each enemy character's behavior are described in the following subsections.

In order for all four enemy characters to follow the player, the method GetClosestDirection is provided. GetClosestDirection allows enemy characters to determine their next direction based on a provided target position. Note that this target position is not necessarily the player's position but has to be related to it in some way - certain ghosts, for example, target the location several tiles ahead and in the current direction of the player character. The method works by calculating the distance between the enemy and target for all valid directions (determined through the Node Controller). A direction is chosen if it has the shortest distance and doesn't make the enemy character backtrack - for example, if it is currently moving towards the right the chosen direction can not be "left". The formula used for calculating distance is given below, where d represents the distance calculated, (x_1, y_1) represents the coordinates of the first point - the enemy character's location - and (x_2, y_2) represents the coordinates of the second point - the target's location:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{3.1}$$

The differing targets for each ghost type adds a level of complexity to their behavior, as it allows these enemies to move in different patterns. Additionally, it gives them a sense of personality, which can be enjoyable for the player to watch.

These patterns, although simple individually, do give rise to complex group behavior. For example, two of the ghosts can seemingly "work together" to corner the player character. Other times, the player may also feel a sense of being "ambushed" by the ghost characters. The variety this provides may give room to replay value and provide an exciting level of challenge for the player depending on their skill level.

3.1.4.1 Red Ghost

The red ghost is based on Blinky, one of the ghost characters from the original Pac-Man video game. It chases the player character consistently and is often seen directly following them in the game. Like the original Pac-Man, this game uses the player's current position as the red ghost's target. The algorithm for its movement is the simplest out of all four ghosts, as all it requires is supplying the GetClosestDirection method with the player character's current position coordinates. The direction obtained from GetClosestDirection is then set as the next direction in the Movement Controller. This is illustrated in Fig. 3.2 [3]

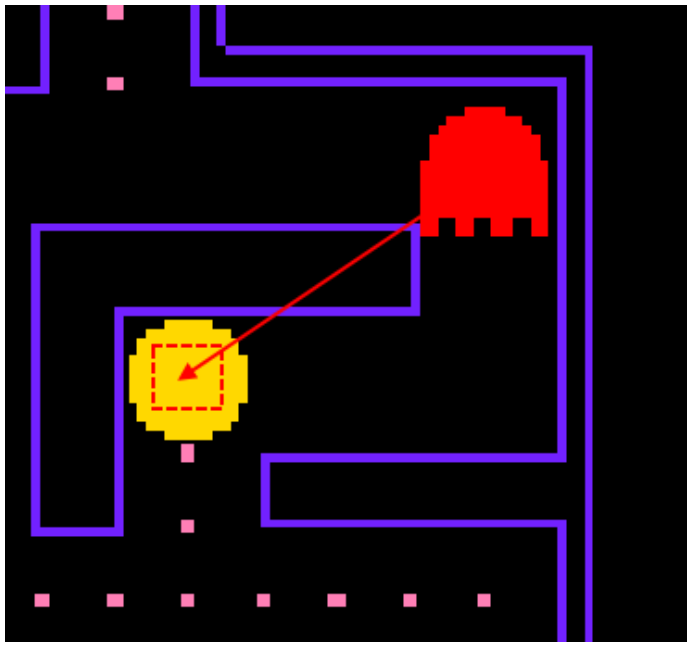

Figure 3.2: Movement pattern for the red ghost type.

3.1.4.2 Pink Ghost

The pink ghost character is based on "Pinky" from the original Pac-Man game. Unlike the red ghost, its target position is two tiles ahead of the player's current position, in the same direction they're heading in. This gives the effect of the ghost "ambushing" the player, as it strives to always constantly move ahead of the player instead of directly tailing them. This is illustrated in Fig. 3.3.

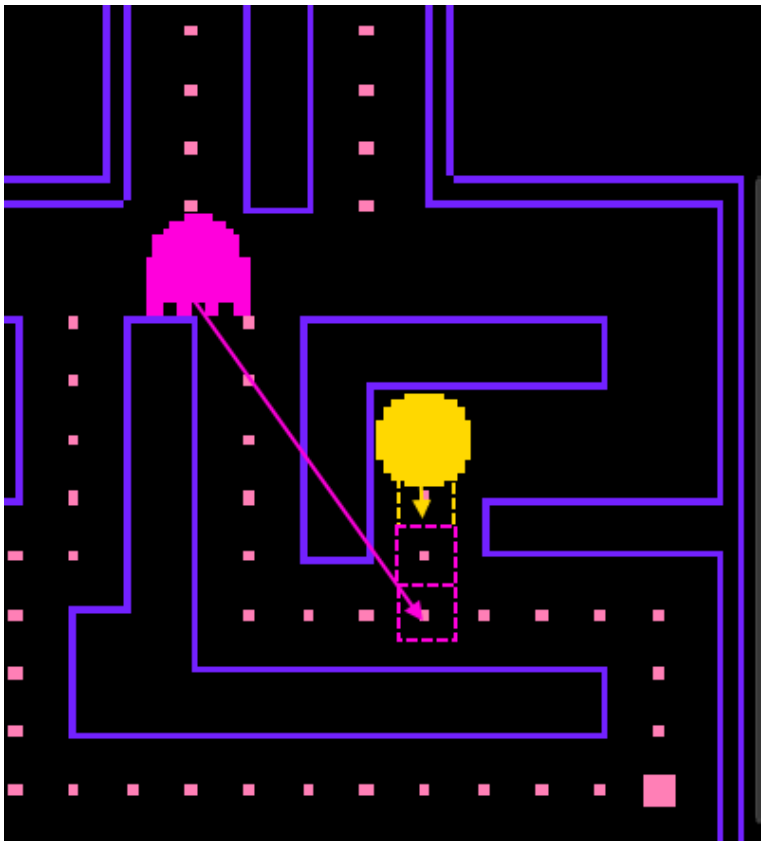

Figure 3.3: Movement pattern for the pink ghost type. The yellow square and arrow illustrates the current direction of the player. The pink arrow shows the target of the ghost.

The algorithm for the pink ghost first involves obtaining the player's target direction and position. Depending on the player's direction, it adds to or subtracts from the x or y coordinate of the player's position a distance of 2 tiles. If the target direction is "left", 2 is subtracted from the player position's x coordinate. Conversely, if the target direction is "right", 2 is added to the player position's x coordinate. Similarly,

for "up" 2 is added to the y coordinate, while for "down" 2 is subtracted from the y coordinate. The resulting position will serve as the target for the ghost.

3.1.4.3 Blue Ghost

The blue ghost character is based on "Inky", who is also from the original Pac-Man game. Inky's character is known in the original game to be difficult to predict. To program this behavior, the blue ghost takes into account the positions of both the player character as well as the red ghost. Its target is calculated firstly through obtaining the location two tiles ahead of the player's current position and in their direction of travel. It then obtains the distances between this location and the red ghost's position in terms of x and y coordinates. These distances are then respectively added to the x and y coordinates of the previously obtained location that is two tiles ahead of the player. Essentially, the algorithm draws a vector from the red ghost to this location and doubles the vector's length in order to find the final target position. This is demonstrated in Fig. 3.4.
In the genetic version of the game, the existence of a red ghost among the genome population is not ensured. Thus, an invisible red ghost is implemented to ensure the blue ghost will always behave unpredictably as planned. This red ghost, unlike its visible counterpart, does not trigger a game over event upon colliding with the player.

3.1.4.4 Orange Ghost

The orange ghost is based on the character "Clyde" from the original Pac-Man game. The character's behavior is unique in the way it switches between two different modes depending on its proximity to the player character. The first mode occurs when the player character is less than eight tiles away from the orange ghost. This mode causes the orange ghost to behave like the red ghost and uses the player's

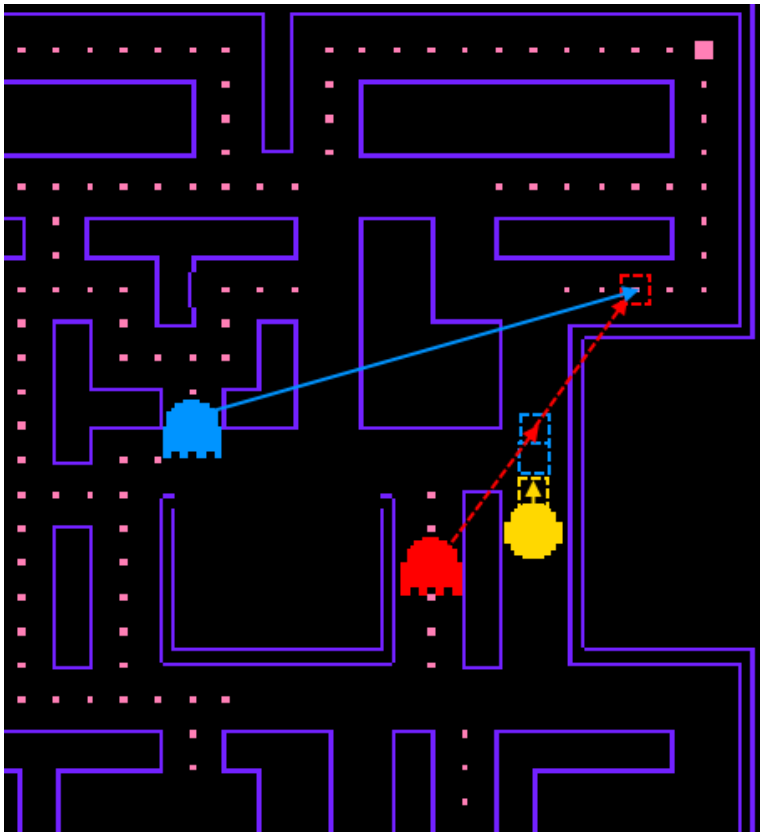

Figure 3.4: Movement pattern for the blue ghost type. The yellow square and arrow illustrate the current direction of the player. The blue arrow shows the ghost's current target.

current position as its target. If the player character is more than eight tiles away, the second mode occurs which sets a tile outside of the maze as its target. This is illustrated in Fig. 3.5.

The algorithm for the orange ghost calculates the distance between it and the player's current position with the same distance formula used in the method GetClosestDirection. As for implementing the second mode, a special node is placed just below the maze's bottom left corner. This node is empty and has no pellet, unlike the ones in the maze.

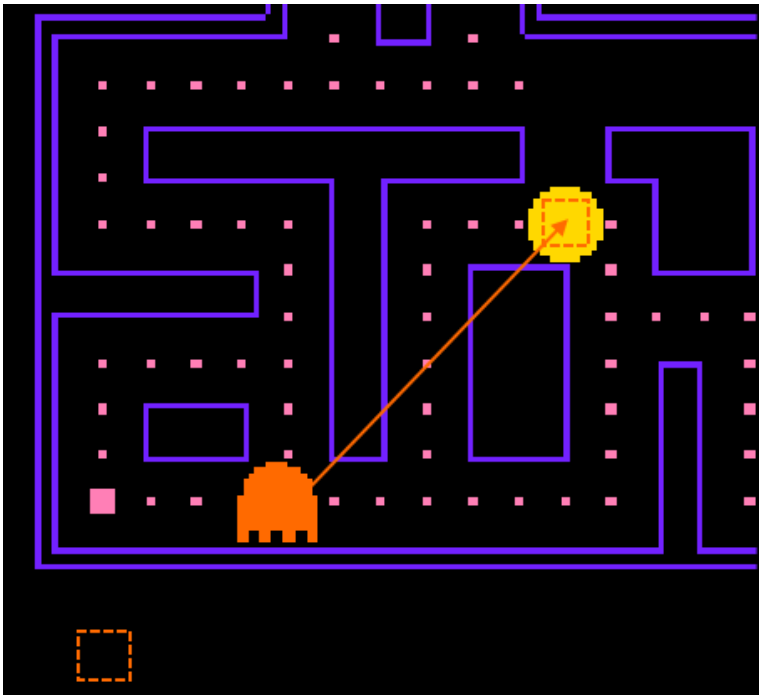

Figure 3.5: Movement pattern for the orange ghost type. Two potential targets exist for the orange ghost - a location outside of the maze and the player character.

3.1.5 Game Manager

The Game Manager controls and keeps track of the various states and events happening in game, such as win or game over scenarios. It also keeps track of the current score and playtime. This is all done through the Game Manager class's Update loop, which constantly updates the time and score as well as number of pellets and energizers in game. If the number of pellets and energizers goes to zero, the win state is set.

Updating score is done by constantly keeping track of the number of pellets as well as the number of energizers available in game. As stated previously in section 3.1, each pellet will reward the player with 1 point while each energizer will give them 10 points. The score is simply the sum of all the pellets or energizers obtained multiplied with their respective points.

It is intended for other classes to reference the Game Manager class for information regarding time, score, and game state. Furthermore, all information in this class is set to public in order for other classes to modify the game state. For example, Enemy Controllers are able to change the game state, which is stored in the Game Manager, to game over if the enemy collides with the player.

3.1.6 HUD Manager

The HUD Manager is in charge of managing the heads-up display of the game. This includes displaying score and time as well as a pop-up that communicates to the player whether they have won or lost the game. As demonstrated in Fig. 3.6, this pop-up appears as a basic grey rectangle accompanied by words saying "You Lost!" or "You Win!" depending on the state the player has achieved after a single playthrough. It also includes a "play again" button which resets the scene and begins another playthrough.

3.2 Genetic Algorithm Overview

The genetic component of the game takes several ideas from Stanley et al.'s NERO game as well as the rtNEAT algorithm, although with the standard binary encoding and mating operators. Firstly, it treats each enemy character on screen as a single genome in the population, and the end of a single playthrough as an epoch or iteration of the genetic loop. This means that the four ghost characters will count as the entire population, and fitness evaluations for each member will only be calculated after a run is over. Secondly, the algorithm allows the player to participate in the evolutionary process and train enemy characters to behave in ways that match the player's skill level - what will now be referred to as the "trainer" model from this point onwards. Training in the case of our game simply constitutes of

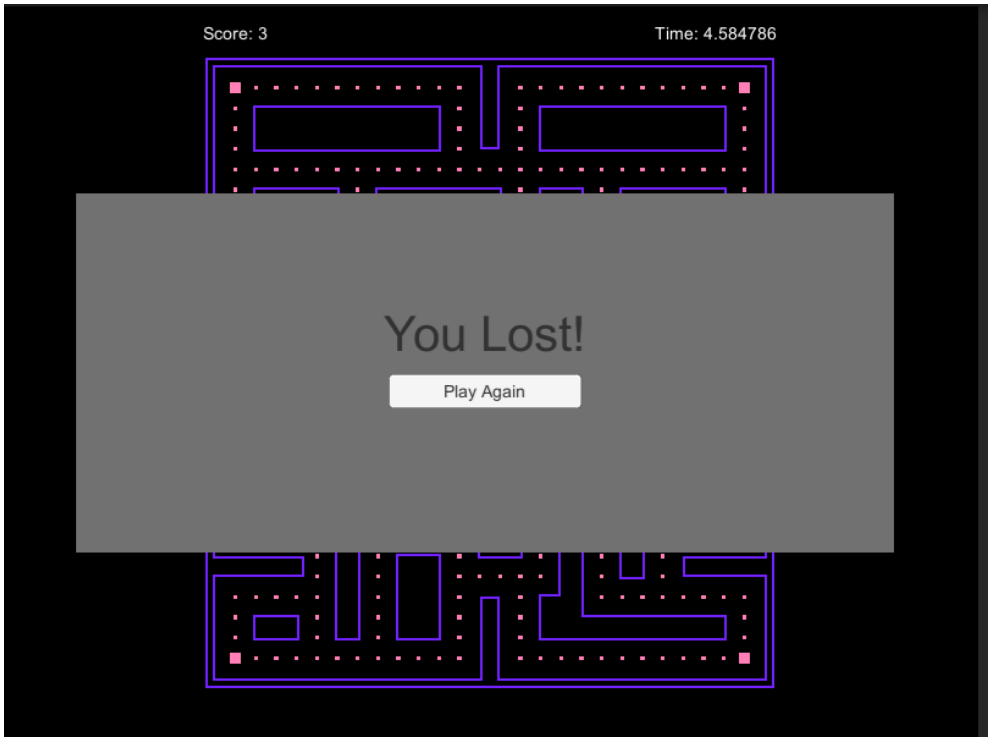

Figure 3.6: Game Over pop-up.

playing several runs against the four enemy characters generated from the current population of genomes. To accommodate evolution, a training session will always consists of more than one run until convergence occurs. Lastly, as a part of the trainer model it is intended for the algorithm to converge to a suitable solution that can be stored for use in future playthroughs. Ideally, multiple solutions from different training sessions can be assembled into a team of new enemies completely tailored to the player's needs - this feature is not implemented in the game however, and is left for future research.

3.2.1 Genome Representation

This genetic algorithm uses binary encoding for its genome representation. It uses a system of 5 bits, with the first two representing ghost types or color, and the last three bits representing speed. The first two bits are capable of representing four

different types - 00 for red, 01 for pink, 10 for blue, and 11 for orange. Meanwhile, the last three bits are capable of representing speeds from 0 to 7, which is a suitable range for enemy characters in game. Note that enemies in the static version of the game have a speed of 3, whereas the player has a speed of 8 in both versions. While the player is always faster than each of the ghosts, these characters can easily overpower the player in terms of numbers, even in slower speeds. With a speed number that is much higher than that of the player's, the game will be difficult to test as loss is almost guaranteed.

The reason as to why both ghost type and speed is taken into account is firstly due to chromosome or genome length. For example, having speed alone would lead to a length of 3. This not ideal for mating operations, as there are not enough possibilities for a good solution to arise [8]. This issue is referred to as lack of design freedom, which does contribute bias to solutions [8]. There is also the observation that even ghosts of the same type can move differently if set to different speeds. This is because each ghost will be responding to the player's positional changes at different locations of the maze despite having the same behavior patterns. Their response will vary regarding the closest direction that would lead each of them to the player. Additionally, speed is a logical and easy way to quantify difficulty. In contrast, a feature such as ghost type would require extensive playtesting to determine their respective difficulty levels. Even then, results obtained from such playtesting may not be applicable to all players. Speed, on the other hand, is more likely to be a universal or objective measure of difficulty. Everyone can agree that faster moving enemies are harder to evade, whereas avoiding slower moving enemies is easier.

3.2.2 Genome Selection

The genetic algorithm implemented for the game encorporates two selection methods. In each iteration of the genetic loop, two genomes are probabilistically selected using the "roulette wheel" algorithm. With roulette wheel, genomes with higher fitnesses have a higher change of being chosen. The pseudocode for this algorithm is provided below in Algorithm 5 [4]. Such genomes are intended to be used for crossover, which produces 2 children genomes. In addition to this selection, after the mating and mutation processes are over, two fittest genomes of the current population are also chosen, a selection process called "elitism". Elitism ensures that the best genomes are always taken to the next generation. In many genetic algorithms roulette wheel and elitism often go hand in hand. The reason for this is because roulette wheel does not guarantee the selection of the best genomes and works probabilistically, which may lead to the loss of good genes in the population pool. Elitism helps mitigate this problem by guaranteeing that those with highest fitness scores will always be selected [4].

Algorithm 5 Roulette Wheel Selection

Select random number between zero and total fitness score
Cumulative sum = 0
for each genome in the population **do**
Add current genome's fitness to the cumulative sum cumulative sum > random numberReturn current genome
end for

Roulette wheel works by first selecting a random number between zero and the total fitness score of the current population. Setting this number aside, the algorithm iterates through the population and performs a cumulative sum calculation at each step. Before moving to the next genome, the algorithms check to see if the current cumulative sum is larger than the random number. If so, the genome at the current index of the population list is selected.

Following the rules of a standard genetic algorithm, moving from one generation to another does not change the population size. Two genomes are removed from the population at each iteration of the genetic loop to be replaced by two new children genomes produced from the crossover and mutation processes.

3.2.3 Mating Operators

For this genetic algorithm, the crossover rate and mutation rate are 0.7 and 0.001 respectively. These values are chosen based on Buckland's recommendations [4]. In general, choosing values for such rates is highly experimental. To reflect evolution in real life, the mutation rate is generally set to be much lower than the crossover rate, as mutations are rare occurrences in nature [4].

The crossover operator chosen for this genetic algorithm is the standard single point crossover operator previous described and explained in chapter 2. Additional pseudocode, however, are provided below in Algorithm 6:

As for the mutation operator, single point mutation is chosen for this genetic algorithm. Similarly, its workings are described in chapter 2. The mutation algorithm iterates through each bit of the genome encoding and probabilistically changes a bit. The pseudocode for this operator is provided below in Algorithm 7:

3.2.4 Fitness Function

The fitness function for this genetic algorithm is formed through decomposition. In this approach, the problem at hand is broken down into several objectives or criteria. Each objective is assigned a weight which determines their importance in terms of a genome's fitness. After a genome's performance is evaluated, scores are assigned to each objective. Such scores are then normalized, multiplied by its weight value and added to form the fitness score.

Algorithm 6 Single Point Crossover

```
function Crossover(mom, dad, child1, child2)
    if randFloat > crossoverRate || mom == dad then    ▷ If the random floating
point number generated is larger than the crossover rate
        child1 = mom
        child2 = dad
    else
        crossPoint = random point to swap
        for index = 0 to crossPoint do
            child1 += mom(index)
        end for
        for index = crossPoint to end of genome bit list do
            child1 += dad(index)
        end for
        for index = 0 to crossPoint do
            child2 += dad(index)
        end for
        for index = crossPoint to end of genome bit list do
            child2 += mom(index)
        end for
    end if
```

Algorithm 7 Single Point Mutation

```
for index = 0 to end of genome bit list genomeBits do
    if randFloat < mutationRate then                    ▷ Flip the bit
        if genomeBits(index) == 0 then
            genomeBits(index) == 1
        end if
        if genomeBits(index) == 1 then
            genomeBits(index) == 0
        end if
    end if
end for=0
```

In this case, four objectives are identified. In the first objective, a genome, or a ghost character, has to strive to collide with the player character and trigger a game over event. However, it should not do this too early in the playthrough. Through some experimentation, the first 6 seconds is chosen as the threshold for this objective, although more research is needed to determine the right value. Points will be rewarded to ghosts that manage to collide with the player, but if they do so before the aforementioned threshold, points will also be deducted. For the second objective, in the case of a win event the genome closest to the player will be rewarded to encourage more aggression. For the third objective, a range is given regarding the average distance between the player and genome throughout the entire run. Genomes that fall outside of this range will be punished, whereas those that fall within the range will be rewarded. The experimental threshold range chosen is between distances 8 and 20 units, keeping in mind that the maximum distance obtainable in the map is about 40.361 units measured between two diagonal corners. Lastly, for the fourth objective, a genome with the speed of zero will be heavily punished in order to remove it entirely from the population. The reason behind this objective is because ghosts with the speed of zero prevents the players from obtaining the last pellet required to win the game.

The mathematical formula provided in Equation 3.2 sums up the fitness function:

$$Fitness(h) = norm_{obj_1}(h) + 1.5 * norm_{obj_2}(h) + norm_{obj_3}(h) + obj_4(h) \tag{3.2}$$

where normalization is done through the min-max normalization method:

$$norm_{obj_i}(h) = \frac{obj_i(h) - min}{max - min} \tag{3.3}$$

h in this case denotes hypothesis, another term for genome. $obj_i(h)$ denotes the score obtained from objective i for genome h. The weights for the four objectives, after

some experimentation, are chosen to be 1, 1.5, 1, and 1 respectively. All objectives, with the exception of objective 4, are normalized, as very large deductions from the fitness is desirable. *min* and *max* represents the minimum and maximum scores obtainable for each objective. Each objective has different minimum and maximum values. For the first objective, the minimum value is 0, and the maximum value is 10. These values are also the minimum and maximum scores for the second objective. For the third objective, the minimum score is -658.902, whereas the maximum score is 36. These scores are the smallest and highest values obtained from a quadratic function within the range of $[0, 40.361]$. Objective 3 utilizes the parabola $y = -(x-8)(x-20)$, where 8 and 20 are the previously mentioned thresholds. The intention behind utilizing such equation is to reward or punish genomes dynamically - for example, distances closer to the range will have less points deducted compared to distances further from the range.

Note that more experimentation is needed to determine the ideal thresholds for this fitness function. For the third objective, the quadratic parabola can also be replaced with other graphs.

In addition to the fitness function, fitness scaling is also performed. In a standard genetic algorithm, scaling is generally used to prevent early convergence from using raw fitness scores [4]. For this genetic algorithm, a method called sigma scaling is chosen, which keeps convergence from happening too slowly or quickly. Its formula is provided below [4].

$$\begin{cases} fitness = 0, & \text{if } \sigma = 0 \\ fitness = \frac{OldFitness - AverageFitness}{2\sigma}, & otherwise \end{cases} \tag{3.4}$$

where σ represents the standard deviation of the population [4]:

$$\sigma = \sqrt{\frac{\sum (f - mf)^2}{N}} \tag{3.5}$$

with f being the fitness of one genome, mf representing the average fitness of the entire population, and N denoting the size of the population.

3.2.5 Implementation

In terms of code, the genetic component of the game is implemented through three classes: GeneticController, GeneticData, and the Genome class. The GeneticController class is where the genetic loop and various mating operators are programmed. In the genetic loop implemented by this class, it considers each run or scene reset in game as one iteration due to the fact that fitness evaluations can only occur after a playthrough is over. The GeneticController is considered to be the main controller behind all evolutionary processes that happen in the software, whereas the two other classes act as helper classes. The first helper class, Genome, encapsulates all information regarding a genome, such as its encoding in the form of a bit list, its fitness score, its age in terms of generation, and access to the ghost GameObject that it instantiated on screen. Throughout all generations, a list of Genome class objects are kept in the GeneticController class as the population. As for the second helper class, the GeneticData class serves to retain information between each run or playthrough in a training session. This class is necessary as resetting a scene in Unity, a feature used to implement the restart or play again button, removes all information collected in the previous playthrough. GeneticData stores game and evolutionary data in static variables, and through its public get and set methods other classes can essentially use it to load and save information.

CHAPTER 4

Conclusion

4.1 Results

It is expected that the population of a genetic algorithm will at one point converge, that is all four ghosts characters on screen will eventually have the same color and speed. With a smaller population, there is no doubt that this will happen relatively quickly within a few generations. The goal is to have the algorithm recognize and remove genes that are deemed unsuitable for the player's experience. This should result in one type of ghost character that is unique to the player and accommodates their level of skill. The easiest scenario to determine is all losses within the first 6 seconds till convergence, which should lead to the genome with the slowest speed no matter what ghost type it belongs to. To recall, speed is an easier way of quantifying difficulty in comparison to ghost types. Although the different ghost behavior patterns do factor into a genome's fitness score, speed also plays a very large role, especially during the initial stages of a run when all ghosts diverge from the same location and are closer to one another. Thus, analysis of results will largely focus on the relationship between a genome's speed and fitness score as well as the end game state (i.e. win or loss).

A test was devised in order to determine whether or not the algorithm will

behave as expected. One part of the test involves simulating a total loss scenario, where the player loses every run early on until convergence. In this scenario, the player simply stays still in their original spawn location and let the ghosts come to them. Meanwhile, the other part of the test involves playing normally but with at least one win playthrough. Losses in this case do not necessarily occur early on in each run. For both parts, 5 training sessions were performed till convergence to make up for the non-deterministic nature of genetic algorithms.

Tables 4.1. 4.2, 4.3-4.4, 4.5, 4.6 display all data collected in 5 training sessions conducted for the total loss scenario. In all tables, the data obtained suggests that, with the exception of speed 0, the algorithm consistently manages to converge to the genome with slowest speed. In cases where genomes with the speed of 0 are present, they are removed from the population pool early on. Convergence is caused by rewarding slower genomes with higher fitness scores. Fig. 4.1 and 4.2, which illustrate trends in the average speed per generation for training sessions 2 and 4, also display an overall decrease in average population speed stemming from faster genomes being replaced by slower ones as training progresses. For most training sessions, convergence occurs quickly with the exception of training session 3 (Table 4.3), which took 13 generations to reach convergence. The average number of generations taken for 5 training sessions of the total loss scenario is 5.2 generations. Overall, for this scenario, the data obtained suggest that the algorithm behaves as expected.

For the second scenario, data from Tables 4.7, 4.8, 4.9, 4.10, 4.11 suggests that the algorithm works as expected even with a combination of loss and win runs within a single training session. While additional playtesting is needed to see whether players find the end results produced by the algorithm to be suitable for their skills, the algorithm does seem to behave in a logical manner. There is a slight positive correlation between higher speed and fitness in win runs. As seen in generation

Generation	Color	Speed	Fitness	ScaledFit	Age
1	2	2	0.93335	0.40999	0
	0	3	0.93373	0.43352	0
	1	4	0.92606	-0.04214	0
	3	6	0.91380	-0.80137	0
2	2	2	0.92732	0.28868	0
	2	2	0.92732	0.28868	0
	0	3	0.90778	-0.86603	1
	2	2	0.92732	0.28868	1
3	2	2	0.89357	0	0
	2	2	0.89357	0	0
	2	2	0.89357	0	1
	2	2	0.89357	0	1

Table 4.1: Training session 1 for total loss scenario

Generation	Color	Speed	Fitness	ScaledFit	Age
1	1	1	0.92981	0.28868	0
	3	6	0.91765	0.28864	0
	2	0	-299.07	-0.86603	0
	2	3	0.93660	0.28871	0
2	1	1	0.93321	0.28868	0
	1	1	0.93321	0.28868	0
	2	3	0.91004	-0.86603	1
	1	1	0.93321	0.28868	1
3	1	1	0.89746	0	0
	1	1	0.89746	0	0
	1	1	0.89746	0	1
	1	1	0.89746	0	1

Table 4.2: Training session 2 for total loss scenario

Generation	Color	Speed	Fitness	ScaledFit	Age
1	0	5	0.89899	0.28863	0
	1	4	0.91506	0.28870	0
	2	4	0.91596	0.28870	0
	3	0	-299.07	-0.86603	0
2	0	5	0.89427	-0.5	0
	0	5	0.89427	-0.5	0
	1	4	0.91249	0.5	1
	2	4	0.91249	0.5	1
3	0	5	0.90230	-0.5	0
	0	5	0.90230	-0.5	0
	1	4	0.91482	0.5	2
	2	4	0.91482	0.5	2
4	0	5	0.90164	-0.5	0
	0	5	0.90164	-0.5	0
	1	4	0.91349	0.5	3
	2	4	0.91349	0.5	3
5	0	5	0.90168	-0.5	0
	0	5	0.90168	-0.5	0
	1	4	0.91436	0.5	4
	2	4	0.91436	0.5	4
6	0	5	0.90952	-0.5	0
	0	5	0.90952	-0.5	0
	1	4	0.91771	0.5	5
	2	4	0.91771	0.5	5
7	0	5	0.90868	-0.5	0
	0	5	0.90868	-0.5	0
	1	4	0.91733	0.5	6
	2	4	0.91733	0.5	6

Table 4.3: Training session 3 for total loss scenario

Generation	Color	Speed	Fitness	ScaledFit	Age
8	0	5	0.89427	-0.5	0
	0	5	0.89427	-0.5	0
	1	4	0.91249	0.5	7
	2	4	0.91249	0.5	7
9	0	5	0.90147	-0.5	0
	0	5	0.90147	-0.5	0
	1	4	0.91313	0.5	8
	2	4	0.91313	0.5	8
10	0	5	0.90952	-0.5	0
	0	5	0.90952	-0.5	0
	1	4	0.91771	0.5	9
	2	4	0.91771	0.5	9
11	0	5	0.89649	-0.5	0
	0	5	0.89649	-0.5	0
	1	4	0.91123	0.5	10
	2	4	0.91123	0.5	10
12	2	4	0.91204	0	0
	2	4	0.91204	0	0
	1	4	0.91204	0	11
	2	4	0.91204	0	11
13	2	4	0.88943	0	0
	2	4	0.88943	0	0
	2	4	0.88943	0	1
	2	4	0.88943	0	1

Table 4.4: Training session 3 for total loss scenario (continued)

Generation	Color	Speed	Fitness	ScaledFit	Age
1	3	7	0.91536	-0.44146	0
	1	3	0.93660	0.83924	0
	1	6	0.91765	-0.303	0
	0	5	0.921107	-0.09478	0
2	1	3	0.93019	0.28868	0
	1	3	0.93019	0.28868	0
	1	3	0.93019	0.28868	1
	0	5	0.91540	-0.86603	1
3	1	3	0.91051	0	0
	1	3	0.91051	0	0
	1	3	0.91051	0	1
	1	3	0.91051	0	1

Table 4.5: Training session 4 for total loss scenario

Generation	Color	Speed	Fitness	ScaledFit	Age
1	1	7	0.91447	0.28865	0
	1	0	-299.07	-0.86603	0
	3	2	0.93184	0.28872	0
	0	7	0.91447	0.28865	0
2	1	7	0.91689	-0.28868	0
	1	7	0.91689	-0.28868	0
	3	2	0.93102	0.86603	1
	1	7	0.91689	-0.28868	1
3	3	2	0.93102	0.28868	0
	3	2	0.93102	0.28868	0
	3	2	0.93102	0.28868	2
	1	7	0.91689	-0.86603	1
4	3	2	0.90222	0	0
	3	2	0.90222	0	0
	3	2	0.90222	0	1
	3	2	0.90222	0	1

Table 4.6: Training session 5 for total loss scenario

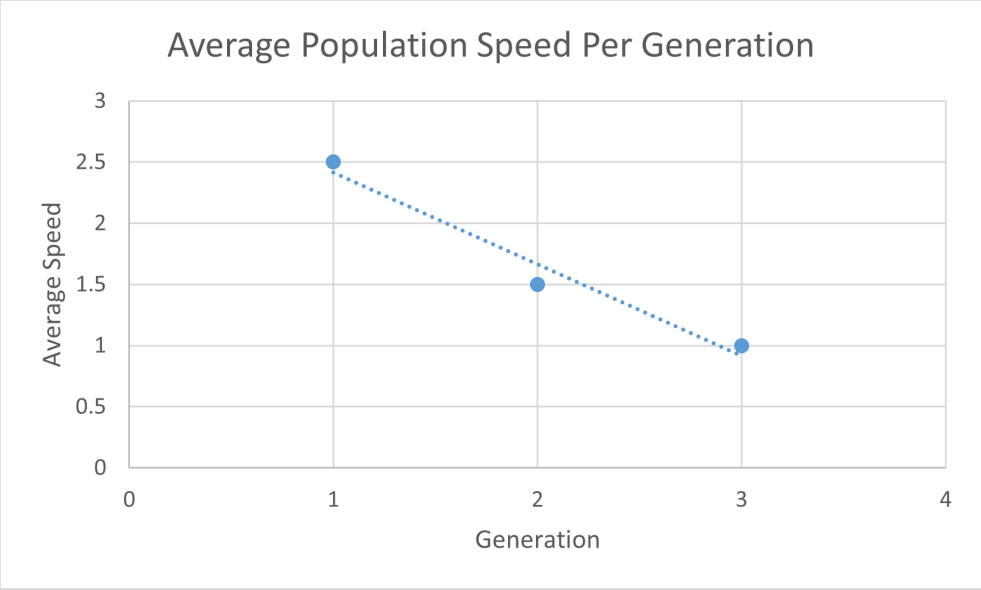

Figure 4.1: Average population speed per generation in a total loss scenario for training session 2.

2 of training session 4 (Table 4.10) and generation 2 of training session 2 (Table 4.8), upon winning the fastest genomes are rewarded with a higher fitness score. However, in generation 2 of training session 3 (Table 4.9) or generation 1 of training

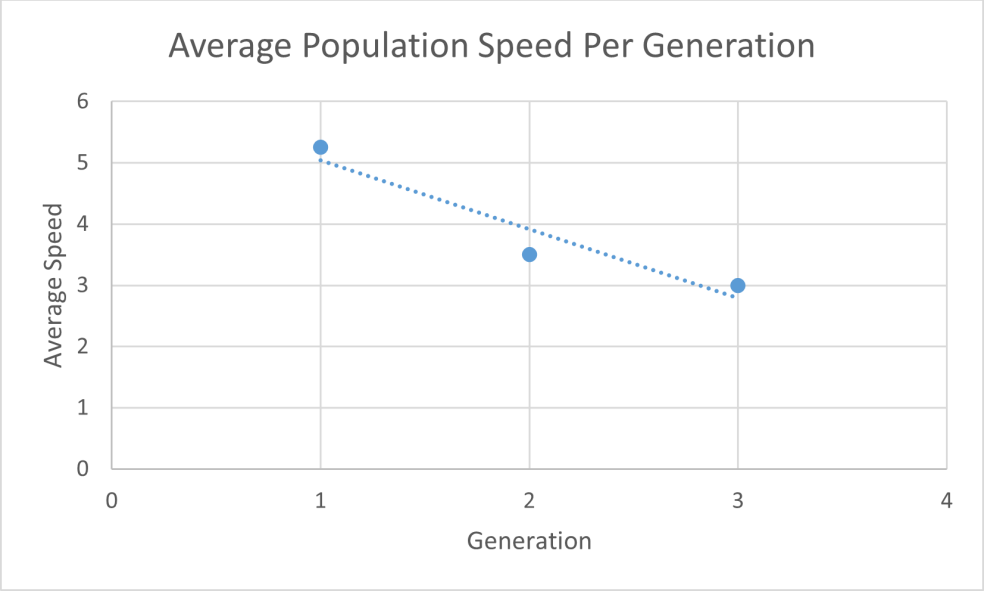

Figure 4.2: Average population speed per generation in a total loss scenario for training session 4.

session 5 (Table 4.11), this is not the case. Observe that in the latter examples, the population is filled with what can be deemed as drastically different ghost types. The orange ghost, for example, which is present in such examples, generally do not venture beyond the bottom left corner of the maze. The red, pink, and blue ghosts on the other hand, tend to travel together all across the maze to ambush the player. Therefore, the red, pink, and blue ghosts have similar behavior. This suggests that the algorithm rewards faster and thus harder genomes for more skilled players if ghost types are similar enough for speed to become the only factor that dictates difficulty. Furthermore, while the difficulty levels of each of the four ghost types are unknown, the orange ghost is more likely to fall outside of the distance threshold range of objective 3 in the fitness function due to its behavior patterns.

Overall, it appears that the algorithm does respond appropriately to different scenarios. However, there is one untested scenario - the total win scenario. This is due to the difficulty of the game exceeding the researcher's skill. Participants with higher skill levels will be needed to obtain data for this case.

Generation	Status	Color	Speed	Fitness	ScaledFit	Age
1	Loss	0	2	0.99905	0.47272	0
		1	2	0.99940	0.48012	0
		3	3	0.96444	-0.26186	0
		0	7	0.94422	-0.69098	0
2	Win	0	2	0.98954	-0.28868	0
		0	2	0.98954	-0.28868	0
		1	2	0.99005	0.86603	1
		0	2	0.98954	-0.28868	1
3	Win	1	2	0.99888	-0.28868	0
		1	2	0.99888	-0.28868	0
		1	2	0.99888	-0.28868	2
		0	2	0.999902	0.86603	1
4	Win	0	2	0.999945	0.28868	0
		0	2	0.99945	0.28868	0
		0	2	0.99945	0.28868	2
		1	2	0.99915	-0.86603	1
5	Win	0	2	0.993178	0	0
		0	2	0.993178	0	0
		0	2	0.993178	0	1
		0	2	0.993178	0	1

Table 4.7: Training session 1 for at normal playthrough scenario

Generation	Status	Color	Speed	Fitness	ScaledFit	Age
1	Loss	0	2	0.95879	-0.70437	0
		2	5	0.98413	0.19445	0
		0	4	0.99738	0.66486	0
		2	6	0.97428	-0.15494	0
2	Win	0	4	0.98277	-0.28868	0
		0	4	0.98277	-0.28868	0
		0	4	0.98277	-0.28868	1
		2	5	0.98848	0.86603	1
3	Loss	0	0	-299.005	-0.28868	0
		0	0	-299.005	-0.28868	0
		2	5	0.95905	0.86603	2
		0	0	-299.005	-0.28868	1
4	Loss	2	5	0.95020	0.28868	0
		2	5	0.95020	0.28868	0
		2	5	0.95020	0.28868	3
		0	0	-299.001	-0.86603	1
5	Loss	2	5	0.98076	0	0
		2	5	0.98076	0	0
		2	5	0.98076	0	1
		2	5	0.98076	0	1

Table 4.8: Training session 2 for at normal playthrough scenario

Generation	Status	Color	Speed	Fitness	ScaledFit	Age
1	Loss	3	2	0.99677	0.51078	0
		1	1	0.99630	0.48634	0
		1	4	0.97617	-0.5508	0
		0	5	0.97820	-0.44632	0
2	Win	3	2	0.97993	-0.28868	0
		3	2	0.97993	-0.28868	0
		3	2	0.97993	-0.28868	1
		1	1	0.98204	0.86603	1
3	Win	3	2	0.99525	0.28868	0
		3	2	0.99525	0.28868	0
		1	1	0.99275	-0.86603	2
		3	2	0.99525	0.28868	1
4	Win	3	2	0.99097	0	0
		3	2	0.99097	0	0
		3	2	0.99097	0	1
		3	2	0.99097	0	1

Table 4.9: Training session 3 for at normal playthrough scenario

Generation	Status	Color	Speed	Fitness	ScaledFit	Age
1	Loss	0	0	-299.069	-0.86603	0
		0	5	0.93765	0.28866	0
		3	3	0.94611	0.28870	0
		3	4	0.941713	0.28868	0
2	Win	3	4	0.99988	0.28868	0
		3	4	0.99988	0.28868	0
		3	3	0.99185	-0.86603	1
		3	4	0.99988	0.28868	1
3	Win	3	4	0.98208	0	0
		3	4	0.98208	0	0
		3	4	0.98208	0	1
		3	4	0.98208	0	1

Table 4.10: Training session 4 for at normal playthrough scenario

Generation	Status	Color	Speed	Fitness	ScaledFit	Age
1	Win	3	4	0.99356	0.76102	0
		0	4	0.97801	-0.44852	0
		3	6	0.98553	0.13602	0
		0	4	0.97801	-0.44852	0
2	Loss	3	4	0.99816	0.28868	0
		3	4	0.99816	0.28868	0
		3	4	0.99816	0.28868	1
		3	6	0.998088	-0.86603	1
3	Win	3	4	0.98758	0	0
		3	4	0.98758	0	0
		3	4	0.98758	0	1
		3	4	0.98758	0	1

Table 4.11: Training session 5 for at normal playthrough scenario

4.2 Discussion

Although genetic algorithms are suitable for complex, nonlinear problems, adapting this technique to real-time gaming scenarios is difficult. It is clear that games, even within the same genre, vary widely and thus require unique algorithms. Genetic algorithms have to be to tailored to the games' unique mechanics. Furthermore, in the "trainer" model inspired by Stanley et al.'s rtNEAT and NERO, the difficulty lies in the fact that genetic algorithms often work with population sizes much larger than the typical NPC count in a game. This is because smaller population sizes are often at risk for early convergence and poor solutions from losing diversity too quickly [**?**]. As the training process is intended to be a major part of the game, repeatedly achieving poor solutions in this stage will definitely lead to player dissatisfaction. At the same time, the process of achieving a good solution can also interfere with the player's experience, especially when the algorithm takes too long to converge. Thus, a key to creating a good genetic algorithm for our game is to balance between solution quality and training speed. Additionally, when incorporating the cycle of genome replacement in a game, gameplay and mechanics also have to be taken into account. To prevent breaking the player's flow, the cycle of replacement can not happen in the middle of a run unless carefully disguised under a game mechanic. Conversely, only performing fitness evaluations at the end of each run may slow down the evolutionary process significantly. The player will have to play several rounds of seemingly arbitrary runs before reaching the end results, which is not ideal. Thus, another factor to consider would be the trade-off between player experience and algorithm performance.

The algorithm developed for this Independent Study does take into account some of the issues discussed in the above paragraph. It performs fitness evaluations at the end of each run, which is the best possible timing in regards to preventing the loss of the flow during gameplay. Although doing so can slow down the

training process, this is not an issue for our game as each playthrough is fast-paced. Furthermore, convergence happens relatively quickly with a small population size, allowing the player to speed through the training process. While this suggests a strong possibility of early convergence, the results obtained thus far seem sufficient enough to compromise in exchange for better player experience.

One flaw of the current "trainer" model lies in the fact that a game like Pac-Man does rely on the behavior of ghost characters as a group. While each ghost alone performs simple, repetitive movements, as a group they are able to form complex, unpredictable behaviors. For example, the behavior patterns of the blue and red ghosts complement each other. When performed simultaneously they can easily "ambush" the player, even at slower speeds. Because a part of the game relies on enemy group dynamics, the "trainer" model which focuses individually on each ghost fails to take into account all factors that affect the game's difficulty. An alternative model to the current "trainer" model would be the "resource allocation" model, where each genome represents different combinations of the four ghost types. However, this model is difficult to apply as the player would have to play a lot more runs for one training session in order to evaluate the fitness of each genome.

During testing, it was suggested by the collected data that the algorithm gives faster genomes higher fitness scores in win runs if the current population contains ghosts with similar behavior patterns. Perhaps the next step of this study would be to remove ghost types from the genome representation and perform tests solely on speed. This would also solve the problem of certain ghost types having more advantages over others in a potentially imbalanced map. Given that length is important in bit list representations, one could simply allow a larger speed range for enemy characters. A very high speed number however makes the game difficult

to test. Another solution would be to employ a different form of representation and discard the bit list altogether.

It is important to examine whether this research is sufficient to be used in the area of game balancing. To recall, chapter 2 discusses Andrade et al.'s three requirements for difficulty balancing methods. Firstly, the algorithm must be able to quickly identify and adapt to the player's skill. Secondly, it must also be able to track the player's progression in an accurate and timely manner. Lastly, it must also be able to provide seamless and believable changes []. Based on the results obtained in the section above, the genetic algorithm developed for this study does seem to fulfill two of these requirements, namely the first and the third. Regarding the first requirement, the algorithm responds to the player's wins or losses immediately and assigns appropriate fitness scores to each of the genomes in the current population. For the third requirement, seamless and believable changes in the case of a Pac-Man inspired game is about making logical changes in each run and maintaining the player's flow. The algorithm fulfills this requirement by performing fitness evaluations at the end of each run and makes changes to the population at the start of a new run. Changes to the population are done logically - for example, if the player has lost the previous round, the game rewards genomes with an "easier" combination of ghost type and speed, the latter often being slower.

The current algorithm is unable to fulfill Andrade et al.'s second requirement on its own without the "case-injection" stage. This is due to the fact that it relies on immediate results. The algorithm can only make decisions by looking at the outcome from the closest previous run. It basically makes adjustments using a process of elimination. This means that genes from previous runs can no longer be retrieved once they are removed from the population. The algorithm removes more and more genes until it reaches convergence where attributes from a single genome now dominate the entire population. Logically, when it comes to tracking

a player's results, the game has to reintroduce some form of difficulty when the player improves and can now play in a harder mode - this would be where the "case-injection" stage comes in to reintroduce diversity into the population.

One implication of the genetic algorithm developed for this Independent Study is that the resulting enemy genome can then be saved into a "seeding pool" or "case bank", similar to that of the NERO game or the CIGAR algorithm. Afterwards, members of the seeding pool can then be injected into both non-GA (static) and GA (dynamic) playthroughs. This is the long term goal for the project, and can certainly be developed and tested in the future. Playtesting can then be done on both the training stage (that was implemented in this study) and the future "case-injection" stage where genomes obtained from training are applied back into the game. This would cover the gap of information regarding player experience and feedback that could not be obtained in this study.

References

1. Gustavo Andrade, Geber Ramalho, Alex Gomes, and Vincent Corruble. Dynamic Game Balancing: an Evaluation of User Satisfaction. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2(1):3–8, September 2021. URL https://ojs.aaai.org/index.php/AIIDE/article/view/18739. 10, 11, 12, 39

2. Christian Arzate and Jorge Ramirez Uresti. Player-centered game ai from a flow perspective: Towards a better understanding of past trends and future directions. *Entertainment Computing*, 20, 02 2017. doi: 10.1016/j.entcom.2017.02.003. ix, 5, 6, 8, 9, 46

3. Chad Birch. Understanding pac-man ghost behavior. https://gameinternals.com/understanding-pac-man-ghost-behavior, December 2010. 55

4. Mat Buckland. *AI Techniques for Game Programming*. 2002. 63, 64, 67

5. Mihaly Csikszentmihalyi. *Flow and the foundations of positive psychology: The collected works of Mihaly Csikszentmihalyi*. 04 2014. ISBN 978-94-017-9087-1. doi: 10.1007/978-94-017-9088-8. ix, 5, 6, 7, 8

6. Talib Hussain and Gordon Vidaver. Flexible and purposeful npc behaviors using real-time genetic control. pages 785 – 792, 01 2006. doi: 10.1109/CEC.2006.1688391. ix, 37, 38, 39, 40, 41, 42, 43

7. Alexander Jaffe, Alex Miller, Erik Andersen, Yun-En Liu, Anna Karlin, and Zoran Popović. Evaluating competitive game balance with restricted play. AIIDE'12, page 26–31. AAAI Press, 2012. 1, 10, 11

8. Il Yong Kim and Olivier de Weck. Variable chromosome length genetic algorithm for progressive refinement in topology optimization. *Structural and Multidisciplinary Optimization*, 29:445–456, 06 2005. doi: 10.1007/s00158-004-0498-5. 62

9. Chris Leger. Automated synthesis and optimization of robot configurations: An evolutionary approach. 02 1970. 18

10. S.J. Louis and J. McDonnell. Learning with case-injected genetic algorithms. *IEEE Transactions on Evolutionary Computation*, 8(4):316–328, 2004. doi: 10.1109/TEVC.2004.823466. ix, 30, 31, 32, 33, 34, 35, 36, 37

11. Jeremy Ludwig and Arthur Farley. A learning infrastructure for improving agent performance and game balance. 01 2007. 10

12. Tom M. Mitchell. *Machine Learning*. McGraw Hill, 1997. ix, 2, 12, 13, 14, 15, 16, 17, 18, 23

13. Colin R. Reeves. Using genetic algorithms with small populations. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 92–99. Morgan Kaufmann, 1993. 46

14. Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002. doi: 10.1162/106365602320169811. 20, 21, 22, 23, 24, 25, 26

15. K.O. Stanley and R. Miikkulainen. Efficient evolution of neural network topologies. In *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600)*, volume 2, pages 1757–1762 vol.2, 2002. doi: 10.1109/CEC.2002.1004508. 19, 24, 26, 27

16. K.O. Stanley, B.D. Bryant, and R. Miikkulainen. Real-time neuroevolution in the nero video game. *IEEE Transactions on Evolutionary Computation*, 9(6):653–668, 2005. doi: 10.1109/TEVC.2005.856210. ix, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29

17. Mohammad Zohaib. Dynamic difficulty adjustment (dda) in computer games: A review. *Advances in Human-Computer Interaction*, 2018, 2018. doi: 10.1155/2018/5681652. 1