Game Balancing using Genetic Algorithms to Generate Player Agents

Kazuko Manabe, Youichiro Miyake

1 Introduction

Quality assurance testing (QA) in game studios has become a major bottleneck with the increasing development time, costs, and complexity of modern games. There is a growing need for effective and scalable automated QA methods to replace or augment traditional manual approaches. Advancements in computation power and AI intelligence have made it possible now to automate most of the simple QA tasks, including game balance, designer specification checks, and even bug detection in simple environments.

In this chapter, we introduce a method developed at Square Enix which leverages state of the art artificial intelligence to help make life easier for our QA departments. By way of illustration, we will discuss our experience reducing the QA cost for the game *Grimms Notes Repage* [Square Enix 18].

For our method, only the fitness equation and decoding step need to be game specific (described in 3.2.2 Fitness and 3.2.6 Decoding), which allows us to generalize the approach to similar games while only changing the game-specific module. The gains from reducing the QA cost meant our developers could spend more time improving other aspects of the game such as design, optimization, graphics, and sound.

2 Grimms Notes Repage

Grimms Notes Repage is an updated re-release of the original game, *Grimms Notes*, and is a real-time action battle game featuring characters from classic fairy tales. The game takes place in a world brought to life by the Story Tellers, where players must defeat the Chaos Tellers who seek to rewrite those original stories and bring misery to the inhabitants of this world. *Grimms Notes*[1] is free-to-play with in-app item purchases that was available on both iOS and Android.

A battle in *Grimms Notes* consists of two parties fighting each other and is won when the boss or all the enemies are defeated and lost when both heroes belonging to the party leader die. A party is composed of multiple characters, each able to equip multiple items such as weapons, accessories, and various outfits. Figure 1 shows the player's party structure, composed of four main characters each with two associated heroes. The game supports an enormous number of possible party combinations with seven main characters, four accessories and 65 skill cores, as well as approximately 80 heroes, 60 weapons (per hero), and three outfits per hero. In addition, some equipment has a rarity level, with higher

[1] We are discussing the version released on July 26th, 2017 and from here on will simply refer to the game as *Grimms Notes*.

rarity levels meaning stronger equipment[1].

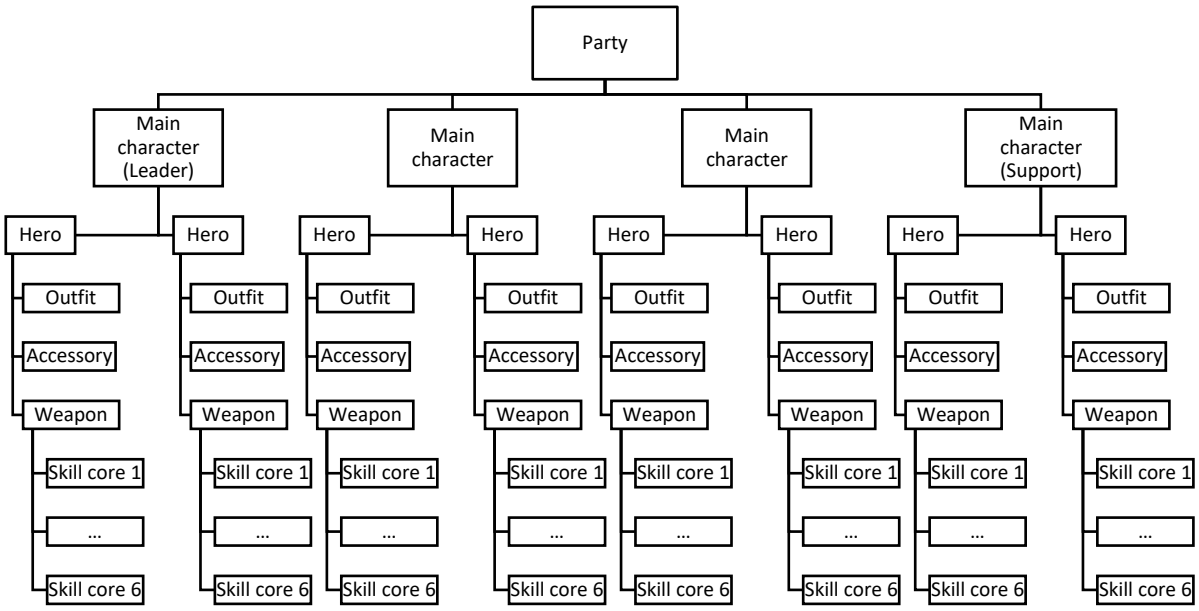

Figure 1: Party structure

The overwhelming variety makes it difficult to manually balance the game to ensure that no combination is overpowered. Making matters more complicated, new items and characters are released every one to two weeks. Each new item requires tuning before release; however, it is impossible to manually test all combinations within this limited timeframe given the combinatorial complexity. Nevertheless, not tuning these items would result in an unacceptable game experience for the player. Thus, we are left with the challenge of how to realistically assure the balance of the game.

3 Our solution

Good game balance in *Grimms Notes* should allow for plenty of strong party combinations while eliminating the overpowered ones. To find such balance-breaking combinations before the release of new characters or equipment, we create virtual player agents in a pre-release version of the game. We then use a genetic algorithm (GA) [Stuart 02] to progress and strengthen those agents; genetic algorithms are well-suited for combinatorial optimization and allow us to visualize the evolution of each party easily. (For other examples of using GA for game balancing, see AiGameDev 12 and Tomobe 16.)

[1] Note that although the game has some party items for sale, players also can acquire the same items through battles without payment. For our discussion, we will assume all possible items are available.

3.1 Genetic Algorithms

A genetic algorithm is a probabilistic algorithm that generates an approximate solution based on Darwin's Theory of Evolution. The main process consists of producing new solutions (offspring) from an existing population and then selecting the fittest among those offspring to produce the subsequent generations. Genetic algorithms have been around for several decades, and there exist many variations.

In this section, we will introduce the specific variant that we used in our research. First, let us define some basic terms:

Gene – *A single parameter of the search space.*
Chromosome – *Collection of genes.*
Individual – *Instantiated chromosome(s), i.e., a solution candidate.*
Population – *Collection of individuals.*
Fitness – *Evaluation function for an individual against the specified problem.*
Crossover – *Process to generate new individuals by breeding two individuals.*
Mutation – *Process to randomly change some genes to escape local optima.*
Generation – *A cycle in the iterative process of the GA (see the algorithm below).*

The general algorithm is then as follows:

1. *Generate an initial, random population.*
2. *Calculate fitness of every generated individual.*
3. *Copy the individual with the highest fitness to the next generation pool.*
4. *With a certain probability, mutate the genes of some individuals in the current population.*
5. *From among this modified population, choose pairs of individuals according to their fitness.*
6. *Generate new individuals for the next generation pool from the chosen parents via crossover.*
7. *If the number of generations reaches a specified number, quit; else, go back to (2).*

Typically, a probabilistic algorithm should iterate until there is no further improvement. For a genetic algorithm, however, it can be difficult to know when or if there will be no further improvement from a subsequent iteration because the random mutations can cause the results to evolve suddenly and by design are impossible to predict. With this in mind, we ran a few thousand generations from various initial populations and found that big improvements almost never occur after about 300 generations. Considering the training time required, we felt it reasonable to terminate each training run after 500 generations, which took about 10 hours.

3.2 Applying a Genetic Algorithm to Grimms Notes

In the following section, we will take a look at our specific approach to game balance in *Grimms Notes* using a genetic algorithm.

3.2.1 Chromosome

As our aim was to uncover overly strong parties, we used the party as our chromosomal unit. The party composition then determined the genes. Specifically, there was a gene for every possible selection within each the seven categories: main character, hero, support hero, outfit, weapon, accessory, and skill core.

4.2 Data statistics

One of our key questions was which genes (main character, hero, weapon, et cetera) result in strong parties. As one of indicators of strength, we used frequency, namely the number of times a gene is used in the winning parties (individuals). The genes included in individuals with high fitness have a high probability of being selected because many of the genes from individuals with high fitness will be carried over to the next generation (3.2.5 Selection). Given that, the frequency of appearance in the winning parties is a useful indicator of the value of that gene.

We executed the GA simulation independently nine times to gather all the data from the winning individuals. Using independent and multiple simulations helped us reduce the influence from local optimums. We calculated the usage frequency of a character as a ratio:

$$(number\ of\ the\ character\ used\ in\ winning\ parties)/(number\ of\ winning\ parties)$$

The frequency values are shown in Table 1. If the genetic algorithm does not work well or all heroes are of equal strength the frequency should be $8 / 300 \cong 0.027$ since there are about 300 heroes, and eight heroes in a party. The highest frequency is 0.71, which is much higher than 0.027, indicating for us that the strength of the characters used in the GA are not uniform and that the algorithm is working appropriately to help us uncover overpowered selections. This table shows the frequency of use of individual heroes, but the same method can be used for combinations with non-hero genes such as weapons.

Table 1: Frequency of use of hero genes included winning individuals

Rank	Frequency	Count	Average of battle duration
1	0.71	138407	533.47
2	0.58	114426	536.66
3	0.45	87945	498.04
4	0.32	62871	517.22
5	0.32	61894	577.96
6	0.24	47642	501.91
7	0.23	46063	539.18
8	0.20	39933	502.56
9	0.20	39411	521.92
10	0.20	39135	505.16

shows the resulting chromosome.

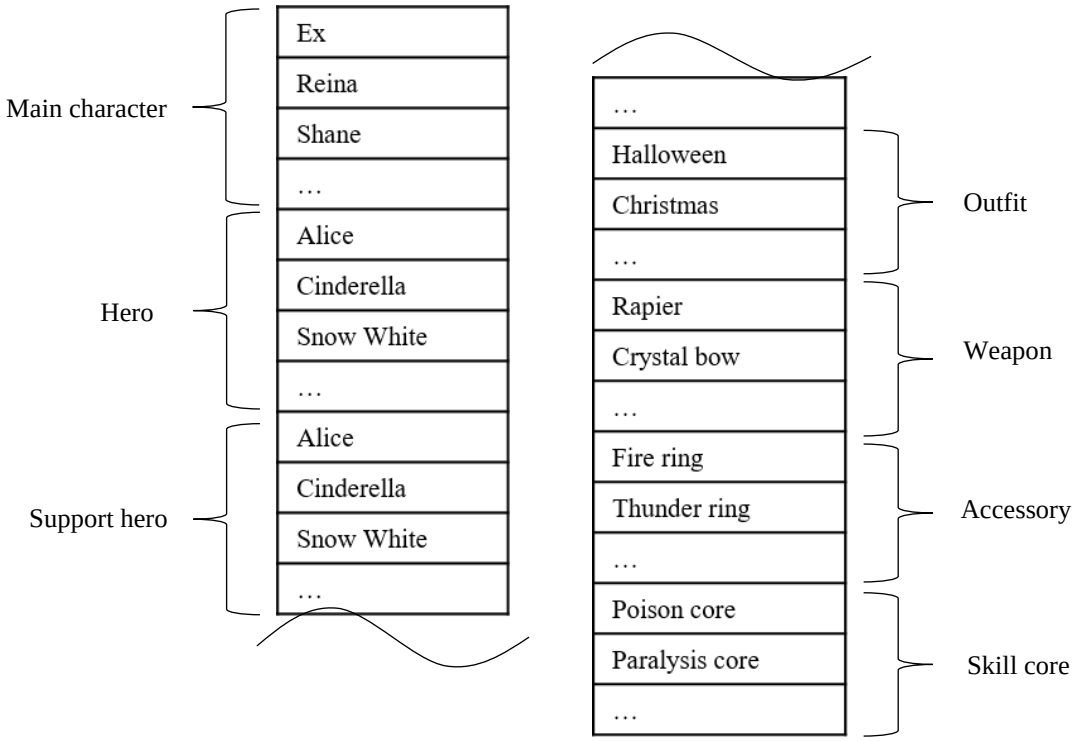

Figure 2: An example of the chromosome

3.2.2 Fitness

Fitness represents the degree of success in an environment according to some measure and the underlying fitness function is a key element in a genetic algorithm. For *Grimms Notes*, the environment is a specified battle, and the fitness represents the strength of the party during that battle.

We explored a few options before arriving at our fitness function. One initial candidate was simply: $(Dealt\ damage)\ /(Received\ damage)$. While reasonable at first glance, the algorithm ended up producing parties that were defeated quickly. Upon investigation, we saw that the algorithm had found that the best way to achieve high fitness is to set heroes with low defense and low vitality as the leader and simply lose as quickly as possible to minimize the received damage. In another instance, we defined the fitness for defeated parties as the length of the battle. This time, the algorithm produced parties that were highly defensive. They were the results of the algorithm selecting for longer battles even if the party ultimately lost. Ultimately, we arrived at the following fitness measure: *When the party wins, the strength is defined by the duration of the battle (how fast was the party able to win). When the party loses, strength is defined by the damage dealt to the enemies.*

It is important to note, therefore, that the appropriate fitness function will depend on the content and context of the game and needs careful consideration. In this case, our function only applies to our specific use case in *Grimms Notes*.

3.2.3 Battle for Fitness

In calculating the party's fitness (the result of the battle), we did not use the actual battle but rather a simulation. In particular, the simulation replaces a few elements of the battle, such as extra skills and the concepts of space and distance, by parameters which can be adjusted by developers familiar with *Grimms Notes*. The biggest advantage of simulating a battle is that the actual in-game battle consumes a few minutes, while this calculation needs only one or two seconds on a standard computer.

3.2.4 Crossover

Crossover is an operation for breeding multiple individuals (parents) by mixing genes in their chromosomes to generate offspring. There are different methods to cross parents. For our game, a crossover is limited to within the same category (Figure 3). In addition, every gene belongs to one of two types: *unique* or *non-unique*. Main character, hero, support hero, and outfit are all *unique* categories in that players cannot have multiple of the same type (e.g. a player cannot have two of the same hero). Weapon, accessories, and skill core, on the other hand, are *non-unique* categories in which players can have more than one of the same type.

We used two well-known methods for our crossovers to preserve the constraints described above: Order Crossover [George 01] for the *unique* categories and Multipoint Crossover [Brian 09] for the *non-unique* categories. Both decide a few points to cut. Order Crossover simply swaps the appearance order of the genes, therefore, the same gene will not be duplicated in their children. Multipoint Crossover swaps a part of the genes between the cut points. As a result, their children can have two or more of the same genes.

3.2.5 Selection

The selection phase selects individuals for crossover according to their fitness. We used a common method called "roulette wheel selection" [Brian 09], which changes the probability of the selection relative to the fitness value. Higher fitness corresponds to a higher probability of selection (Figure 4). The number of elements of one generation is n, the fitness for individuals is $f_k(\mathrm{k} = 1..\mathrm{n})$ and the probability of being selected as a parent for an individual i $(1 \leq i \leq n)$ is $P_i = f_i / \sum_{k=1}^{n} f_k$.

There is a drawback to this method: an individual with a very high fitness will always result in a very high probability of selection. Thus, the next generation will be mostly generated from this individual, leading to a local optimum. To avoid this, common practice is to adjust the fitness value between generations.

In our method we define the fitness function by the following polynomial:

$$f_i^b = c_1^b * d_i^b - c_2^b * t_i^b + c_3^b$$

This represents the fitness for an individual i against battle b, where d_i^b is the damage that the individual i dealt in battle b, t_i^b is the time consumed in battle b of the individual i, c_1^b and c_2^b are arbitrary constants to decide the degree of impact of d_i^b and t_i^b, respectively. Although there may be other polynomial expressions to represent the fitness, any such expression must contain constant numbers to adjust the selection probability for the battle.

Finally, c_3^b is an arbitrary constant to decrease or increase the probability of selection to avoid the problem of a dominant individual.

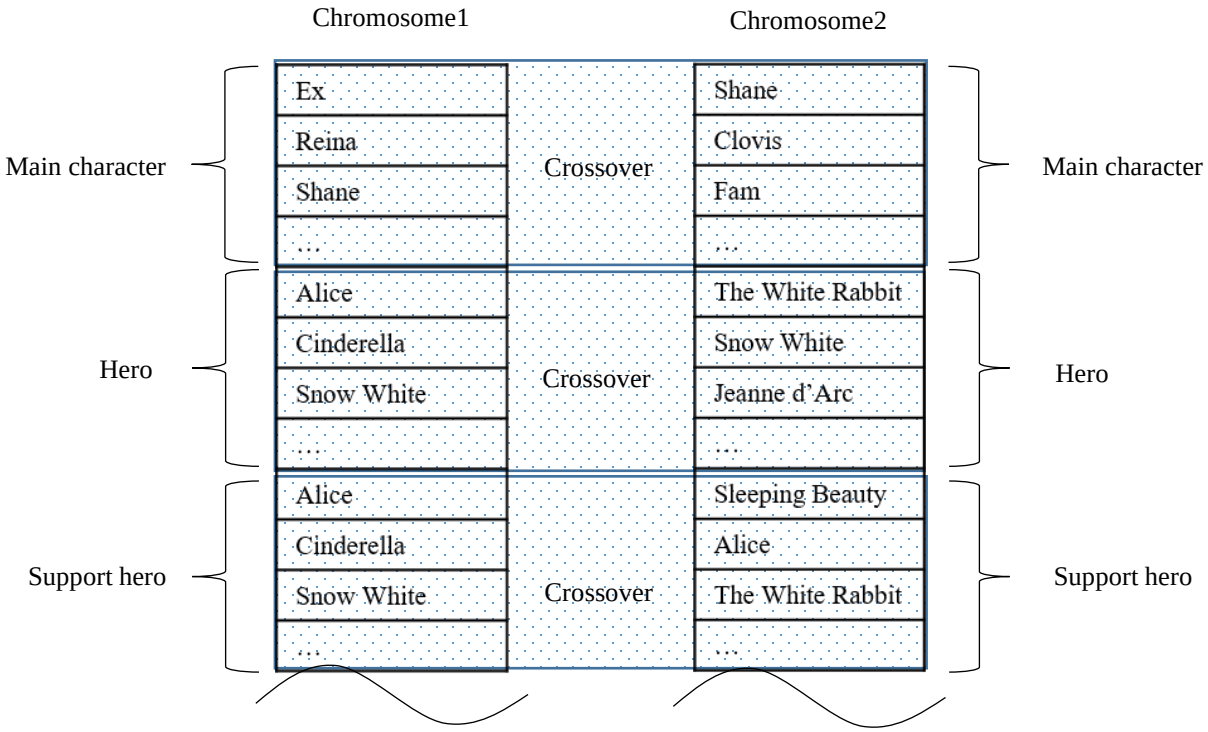

Figure 3: Crossover within the same category

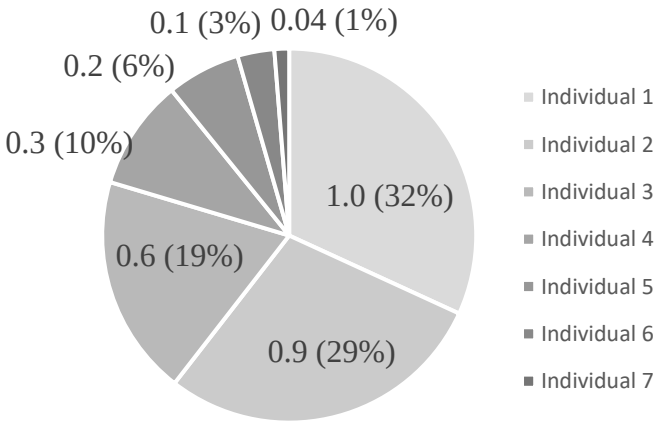

Figure 4: An example of fitness and their selection probability in roulette wheel selection

Given that there are hundreds of kinds of battles in *Grimms Notes* which have different numbers of enemies each with different strengths, it is not realistic to determine the constant values for each battle. Instead, to further avoid local optima as well as avoid spending too much time adjusting the constants, the damage dealt, d_i^b, and the duration of the battle, t_i^b, are mapped to a part of the fitness range as outlined in Figure 5.

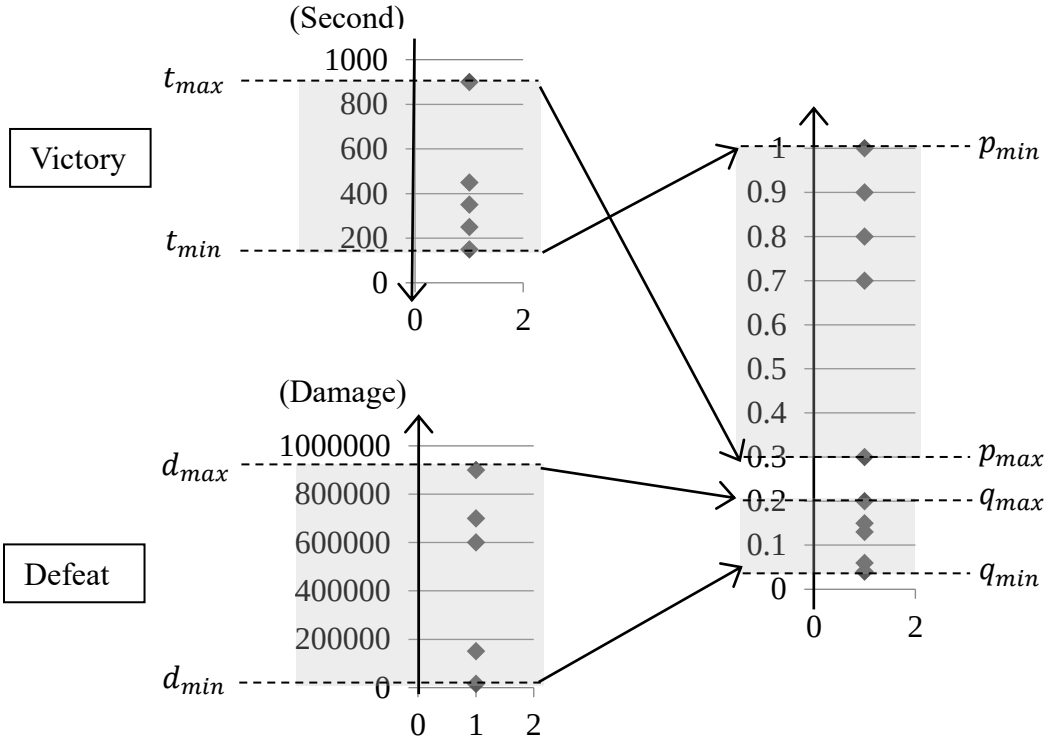

Figure 5: An example of the fitness mapping

Specifically, $t_{min}$is the minimum time, and t_{max} is the maximum time in all winning battles in a generation. The durations of all winning battles are mapped onto $[p_{min}, p_{max}]$ ($p_{min} \neq p_{max}$) on the fitness axis. Fitness f_i in $[p_{min}, p_{max}]$ has the same relation as t_i in $[t_{min}, t_{max}]$. This relation is expressed by the following equation (some exceptions, such as zero division are omitted):

$$\frac{p_{max} - f_i}{f_i - p_{min}} = \frac{t_{max} - t_i}{t_i - t_{min}}$$

$$f_i = \frac{(t_{max} - t_i)p_{min} + (t_i - t_{min})p_{max}}{t_{max} - t_{min}},$$

In the same way, the fitness function in the losing case is expressed as follows:

$$\frac{q_{max} - f_i}{f_i - q_{min}} = \frac{d_{max} - d_i}{d_i - d_{min}}$$

$$f_i = \frac{(d_{max} - d_i)q_{min} + (d_i - d_{min})q_{max}}{d_{max} - d_{min}},$$

Here d_{min} is the minimum dealt damage, d_{max} is the maximum dealt damage in all losing battles in a generation, and all dealt damage in losing battles are mapped on $[q_{min,}\ q_{max}]$ ($q_{min} \neq q_{max}$) on the fitness axis.

The mapping reduces the adjustment of the constants to only four parameters for all battles: $p_{min,}\ p_{max,}\ q_{min,}\ q_{max}$. Currently we set $p_{min} = 1.0$, p_{max} = 0.3, q_{min}= 0.04, q_{max} = 0.2. Note that p_{min} is bigger than p_{max} because a faster time to win means higher fitness. The broader the range for fitness in the winning case than compared to the losing case, the greater the focus on the content of the winning battle; in other words, the difference in duration for the winning battle is treated as a bigger effect than the difference of damage dealt in the losing battle. In *Grimms Notes*, it is very common for players to win their battles and the battle duration is one of most important guidelines to adjust difficulty of the game.

3.2.6 Decoding

The conversion from chromosomes to game-objects is called decoding. The process should be deterministic, meaning a party decoded from the same chromosomes is always the same. The decoding of the chromosomes in our project uses a greedy-like approach. It searches and sets party elements from the upper layer of Figure 1 in the following order: main character, hero, outfit, accessory, weapon, skill core, et cetera.

The biggest advantage of our chromosome structure and the decoding process is its simplicity. While there may be more suitable chromosome structures for *Grimms Notes* that would make our algorithm more efficient, such structures may not be applicable to other games, lessening the flexibility of our approach. The simple structure allows us to not care about the content of these updates and the kinds of battles. A re-execution of the algorithm will result in getting agents which will meet the specification of this new environment, which could be a change of hero, a change of battle specification and so on.

Another possible approach is genetic programming, which uses a tree structure for the chromosome instead of a one-dimensional array and searches for an optimal program that when executed will produce a solution. In this case the party of *Grimms Notes* would be a tree structure and the crossover would then exchange their subtrees. This is equivalent to swapping characters equipped with a particular weapon, for example, to plan a party composition. Although this approach may seem more natural, this is only because it is easy to envision the parties in *Grimms Notes* as a tree. This does not necessarily apply to other games or to other problems, thus limiting its generalizability. At the expense of versatility, genetic programming offered us only minimally better results for the game [Manabe 19]. Therefore, which structure ultimately depends on the size of the game project and the number of the other relevant game projects in the company. One-dimensional arrays proved easier to adapt to the problem than tree structures.

3.2.7 Avoiding local optima

Our system for mutation described above can help reduce the chance of a local optimum, but it is not perfect. One simple and effective method to further reduce this risk is to evolve

multiple populations independently, also known as the island model [Kanakubo 21]. For our purposes, we created nine "islands" and evolved the population in each of them. As a result, the nine islands evolved into three different types: parties that focus on attack, parties that focus on not taking damage, and parties that focus on recovery. The difference of the battle duration between them is about 10%. The final step then tries to find a global optimum via immigration, which mixes the island populations. Our exploration so far of this method has yielded some initially promising results, though further study is needed.

4 Results

For *Grimms Notes*, we configured our battle to only have normal enemies instead of special, strong enemies, such as bosses that can appear in the game. This is because the normal battles account for more than 90% of all battles in the game and the bosses have powerful personalities that are not appropriate for the first experiment.

In normal battles, the durations tend to be long because the battles have many enemies. Therefore, the attack, defense and healing effectiveness must all be well balanced for the parties to endure long-term battles. The victory condition is to wipe out all enemies and the defeat condition is that both heroes of the leader are knocked out.

4.1 Simulation

Our GA simulation is done by generating 50 individuals per generation and 500 generations. The number of generations is set so that the simulation can finish within about 10 hours on our computers (Intel® Core™ i7-8700K CPU 3.70GHz, 32GB RAM and others with equivalent specifications).

We experimented with three types of simulations. First, we executed the genetic algorithm with all characters and equipment. For our second simulation we excluded characters and equipment with lower rarities. Finally, to test the magnitudes of the effect of the genetic algorithm, we ran simulations without the genetic algorithm. For all the simulations, we set the number of islands to nine. We showed the best of the results in Figure 6.

Simulation 1 This simulation uses all categories of characters and items. Figure 6 shows the change of the minimum duration of the winning battles. Although all individuals in the first generations typically lose quickly, the battle durations rise sharply up to the 17^{th} generation, where the first winning individual appears. After the appearance of the winning individual the time to win shortens significantly from 816.5 seconds in the 17^{th} generation to 571 seconds in the 494^{th} generations, or about 30%.

Simulation 2 It was found that the evolved individuals in Simulation 1 resulted in the inclusion of some low rarity (common) characters and equipment, which is a problem since we're trying to make strong parties. We removed these clearly weaker characters and equipment components and applied our algorithm again in Simulation 2. The results show greater damage in earlier generations when compared to Simulation 1, and the shortest battle

time to win reaches 467.75 seconds in the 359th generation, which is about a 20% improvement over Simulation 1. Figure 6 shows the results of removing low rarity items.

Simulation 3 To measure the magnitude of the effect of the genetic algorithm in the results in Simulation 1 and 2, we generated 25,000 individuals, which is the same number of individuals in Simulation 1 and 2 (50 individuals multiplied by 500 generations) and let them fight a battle. We limited their characters and equipment to only those with high rarity. This yielded the shortest time to win of 600.5 seconds. The second and third shortest times were 644.25 seconds and 671 seconds, respectively. These results are worse than the ones from the Simulations 1 and 2. The difference of the time shows the magnitude of the effect of our GA. The components of a party with the shortest time to win make it possible for game developers to quickly find the existence of a disproportionately strong party combination before releasing new equipment and characters.

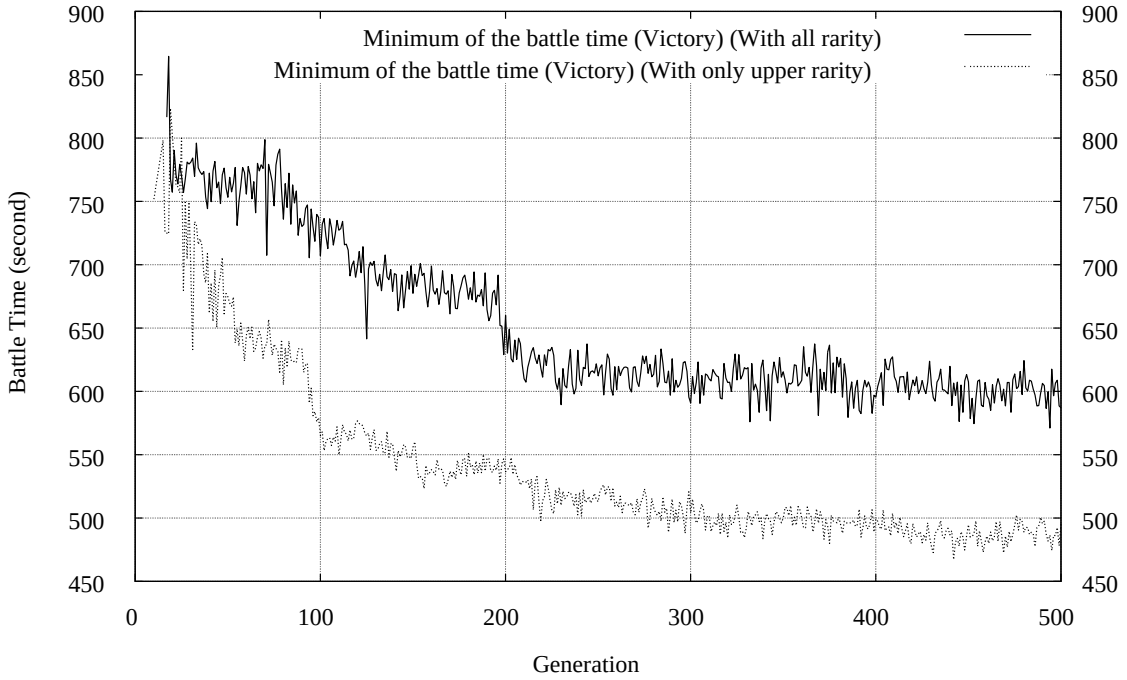

Figure 6: Change of battle time. The horizontal axis represents the generation number, and the vertical axis represents the time (seconds).

4.2 Data statistics

One of our key questions was which genes (main character, hero, weapon, et cetera) result in strong parties. As one of indicators of strength, we used frequency, namely the number of times a gene is used in the winning parties (individuals). The genes included in individuals with high fitness have a high probability of being selected because many of the genes from individuals with high fitness will be carried over to the next generation (3.2.5 Selection). Given that, the frequency of appearance in the winning parties is a useful indicator of the value of that gene.

We executed the GA simulation independently nine times to gather all the data from the winning individuals. Using independent and multiple simulations helped us reduce the influence from local optimums. We calculated the usage frequency of a character as a ratio:

$$(number\ of\ the\ character\ used\ in\ winning\ parties)/(number\ of\ winning\ parties)$$

The frequency values are shown in Table 1. If the genetic algorithm does not work well or all heroes are of equal strength the frequency should be $8 / 300 \cong 0.027$ since there are about 300 heroes, and eight heroes in a party. The highest frequency is 0.71, which is much higher than 0.027, indicating for us that the strength of the characters used in the GA are not uniform and that the algorithm is working appropriately to help us uncover overpowered selections. This table shows the frequency of use of individual heroes, but the same method can be used for combinations with non-hero genes such as weapons.

Table 1: Frequency of use of hero genes included winning individuals

Rank	Frequency	Count	Average of battle duration
1	0.71	138407	533.47
2	0.58	114426	536.66
3	0.45	87945	498.04
4	0.32	62871	517.22
5	0.32	61894	577.96
6	0.24	47642	501.91
7	0.23	46063	539.18
8	0.20	39933	502.56
9	0.20	39411	521.92
10	0.20	39135	505.16

5 Discussion and Conclusion

5.1 Discussion

Our method can not only be applied to *Grimms Notes* but also to other games and game-balancing tasks that can be expressed in terms of genes, chromosomes, and with an appropriate fitness function. We used 500 for the population size and nine for the number of islands, but the appropriate values will vary depending on the complexity of the problem, available computer resources, and time available in the development workflow. It is a good idea to experiment beforehand when you have enough time and can better estimate how many resources you will use.

Also, it bears emphasizing that the design of your fitness function is very important. In our case, we used the time spent in battles for formulating the fitness function because the game used the time a player spends in a battle as a measure of difficulty. A discussion with the game designer is a great place to start.

Visualizing the output of GA is another challenge. In our case, we found that our GA output so many candidates with high fitness values and potentially overpowered elements

that it was not realistic to check all of them by hand. Eventually, though, you will want the game designers to investigate chromosomes according to the visualized data, so narrowing down the number of candidates is important. Our future efforts will be focused on improving visualization and the application of this method to different battles in this and other games. We showed the part of our results in the experiment using genetic programming [Manabe 19], however, we must continue to try to improve how to show the data.

5.2 Conclusion

Like many games, the high complexity of *Grimms Notes*, combined with frequent updates, makes manual approaches to QA and specifically game balance infeasible. The purpose of our work was to find a way to automate the process of identifying disproportionately strong party combinations in the game that could suggest imbalanced items or abilities before they are introduced to players. To do so we applied a genetic algorithm, leveraging party strength as a measure of fitness to efficiently identify outliers. Previous to this approach, our game designers were required to judge whether the party compositions would imbalance the game by checking as many party candidates as possible by hand. By introducing our GA method, however, they can now limit their searches to only the strongest party candidates identified by the GA, significantly reducing the time and therefore the QA cost to balance the game.

6 References

[AiGameDev 12] AiGameDev.com. 2012. Making Designers Obsolete? Evolution in Game Design, http://aigamedev.com/open/interview/evolution-in-cityconquest/ (accessed February 12, 2018).

[Brian 09] Brian Schwab: AI Game Engine Programming, Second Edition, Charles River Media, 2009.

[George 01] George F. Luger: ARTIFICIAL INTELLIGENCE - Structures and Strategies for Complex Problem Solving, Addison Wealey, 2001.

[Kanakubo 21] Masaaki Kanakubo, Shima moderu to MGG moderu (Island model and MGG model), https://www.sist.ac.jp/~kanakubo/research/evolutionary_computing/island_mgg.html (accessed January 28, 2021).

[Manabe 19] Kazuko Manabe, Balancing Nightmares: An AI Approach to Balance Games with Overwhelming Amounts of Data, Game Developers Conference 2019.

[Square Enix 18] SQUARE ENIX CO., LTD.: Gurimu nôtsu (Grimms Notes) Repage | SQUARE ENIX, http://www.grimmsnotes.jp/ (accessed February 12, 2018).

[Stuart 02] Stuart Russell, Peter Norvig: Artificial Intelligence - A Modern Approach (2nd

Edition), Prentice Hall, 2002.

[Tomobe 16] Hironori Tomobe, Toyokazu Handa: AI ni yoru gêmu apuri unyô no kadai kaiketsu eno apurôti (An approach for solving problems of game management by AI), CEDEC2016, http://cedil.cesa.or.jp/cedil_sessions/view/1511 (2018-1-31 referred).