Lessons from Testing an Evolutionary Automated Game Balancer in Industry

Mihail Morosan
University of Essex
Colchester, UK
mmoros@essex.ac.uk

Riccardo Poli
University of Essex
Colchester, UK
rpoli@essex.ac.uk

***Abstract*—Game balancing has been an important area of academic research in the past few years, with various methods of approaching the task being proposed. At this point in time, however, industry impact has been minimal, with these approaches appearing overwhelming or expensive to game designers and developers. The work presented in this paper takes one of the algorithms and approaches previously researched and, in cooperation with a commercial games studio, defines a specification language and tests its applicability in a real world scenario. The game being balanced in this paper is of a different genre to those previously targeted by the algorithm, with vastly different mechanics and expectations of what is considered a balanced state. Results indicate both great potential of the research area in general, but also highlight challenges to achieving mainstream use in the real world of balancing techniques.**

***Index Terms*—Genetic algorithm, industry, balancing, games**

I. Introduction

A. Motivation

Great amounts of research is done in games, however there is a disconnect between what is valuable to researchers and what is valuable to the industry [1]. Algorithms developed in academia are often only applied on synthetic problems, with little time from the researchers available for testing in real games. Similarly, most game professionals do not have the time or resources to invest in understanding the research and applying it to their games. The exceptions are usually the large game corporations [2].

Leaving the task of defining the success of an algorithm in the hands of the experiment designer brings both advantages and disadvantages. The immediate advantage is preserving creative freedom, something very valuable to an artistic medium such as games. Not all games are created equal, and not all games have the same definitions of "fun" or "engaging". However, two immediate disadvantages are the requirement that the algorithm be able to adapt to any and all goals the designer has put in place, alongside the challenge of presenting the system's limitations in an easy to understand manner. Obtuse systems are less likely to be adopted by game developers.

In this paper we briefly propose a method of communicating requirements and results between algorithms and game developers, as well as an experiment run with it. We used the algorithms defined by Morosan and Poli [3], applying them to the commercial game *ComPet*.

B. Previous work

"Balance" is a very broad umbrella term for many aspects of game design. The most generic definition, supplied by Schreiber [4], is that game balance "is mostly about figuring out what numbers to use in a game". This can cover procedural content generation [5], parameter optimisation in games, generating new games entirely [6], and potentially more. This work looks at balance as the process of taking existing elements and mechanics in a given game, alongside their designers' requirements, then exploring the search space of allowed changes to fit them.

There has been significant research done on using a designer's requirements for a balanced game and achieving those goals through the use of computational intelligence.

Mahlmann *et al.* [7] worked on matching game elements from Dominion, a popular board game, to create interesting variations of it for players. They successfully used GAs and AI agents to essentially create new games in the Dominion design space. There was no information on whether this approach was applied in any commercial manner.

Work has also been done on balancing the mechanics available in a game of *Top Trumps* in regards to current literature understanding of fairness and excitement [8]. The approach can be useful when designers are uncertain as to what they desire and are willing to accept an academic understanding of "fun". However it might fall flat should that understanding of a game's requirements differ from the designer's vision. Similarly to the previous research presented, no information was given on whether this was adopted by the designers of *Top Trumps*.

Beyer *et al.* present an integrated process for game balancing [9], describing challenges and potential avenues for success, but it comes from the angle of academic research. It does not go in depth on how important a designer's input is and how much can change in the experimentation phase, as well as the structure of both small and large game studios and their internal pipeline for development.

Work by Morosan and Poli achieved designer defined balance requirements using games of *Ms PacMan* and *StarCraft* by evolving changes to game variables using genetic algorithms (GA) [3]. The requirements varied from influencing a strategy's effectiveness to altering the game's difficulty.

978-1-5386-6304-2/18/$31.00 ©2018 IEEE

Satisfactory configurations are to be defined by the designers of the game, which can include anything they deem important, from the length of a level to the behaviours exhibited by its players. By using the same method of defining goals, Morosan and Poli also balanced the behaviour of artificial agents playing *Ms PacMan* [10]. There was also no mention of commercial success using these approaches.

Chen *et al.* [11] applied GAs to find balanced character skills for role-playing games. It was done on a very small scale, with a minimal custom game model used and a simple rules-based agent controlling behaviour. The approach does not consider how their methodology would apply to real games or the presence of noise generated by humans or AI agents. As a result, it is unclear how it would behave in a real-world scenario.

II. Methodology

A. Introduction

After cooperation with a commercial games studio, MindArk Sweden, we believe we have built the foundation for a simple, yet robust, language for game balance specifications.

This language allows designers to define what can be changed in their games, as well as what to evaluate when calculating the success of any changes, regardless of the game.

Decoupling the algorithms to balance the games from the games themselves can open up research significantly. When someone writes a specification file for their game, they open it up to many other researchers' algorithms, potentially offering them better results.

This does require a small amount of extra work from the developers of a game, as said game needs to understand how to actually apply any changes it is given by an algorithm. It is significantly easier when having access to the source code and being familiar with it.

The specification language is, essentially, a structured JSON file, making it really easy to read by both humans and software programs, while being easy to manipulate.

This language was inspired by work done by groups such as Metaheuristics in the Large [12] and work by Swan *et al* [13]. Many of the tools and communication protocols described in this work would be easily adapted to other approaches to a more standardised method of algorithm selection and design.

B. ComPet

ComPet is a commercial turn-based strategy game developed by MindArk [14]. It gives the player control over a roster of characters called pets, each with its own attributes (such as initiative or endurance), its own experience level and various combat abilities.

The game has two main gameplay loops. The default one is where the player battles the various beasts in the game's campaign mode in one-on-one combat with one of their own pets (see Figure 1). This campaign mode is increasingly more difficult and requires the player to adapt to different strategies or even wait until their pets are strong enough to handle the challenge.

Fig. 1: Screenshot from *ComPet*'s battle mode

TABLE I: *ComPet* beasts in the experiment gauntlet

Name	Level	Health	Ferocity	Endurance
Angel	1	21	0	15
Starbright	1	24	15	25
Jethro	2	29	25	15
Ricktick	3	41	15	20
Forage	3	46	30	6
Harold	4	48	32	30
Fierce Frank	5	45	70	12
Thomas Short-Tail	5	58	15	20
Brutal Bill	6	80	15	50

Battling other players in 1-on-1 combat is the second gameplay loop. This mode has two goals: gathering experience points to strengthen the player's pets, and becoming a better ranked player on the leaderboards.

For this research, the game's campaign mode was abstracted into the concept of a 'gauntlet'. A gauntlet represents a series of beasts to be fought, as well as a single pet the player starts with. The role of the player is taken by an AI agent developed by MindArk.

This AI agent is rules-based with rules derived from real-world play tactics. It behaves admirably given the environment and is an acceptable approximation of human players for the purposes of this experiment.

To simulate actual gameplay, one would start a fight with the first beast in the series. If victorious, the player's pet is rewarded with experience points, levelling them up if enough points were collected. Then, the pet would attempt to fight the next beast in the series. If unsuccessful in a fight, the player would focus on the most recently defeated pet (or the first one if that is impossible) for several fights, hoping to gather enough new abilities and attribute points to try again.

Once the pet successfully defeats the final beast in the series (or a maximum number of fights has been reached), all the relevant metrics from every single fight are collected. These metrics could be used either for manual analysis by a designer, or for automated design methods, such as those presented in this work.

While the gauntlet is a condensation of the game's campaign mode, the combat between pets and beasts is based on the

TABLE II: *ComPet* metrics collected on the gauntlet playing the unchanged version of the game

Name	Defeated After # Attempts	Wins Per Run
Angel	1	1
Starbright	1.5	1
Jethro	10	1
Ricktick	20	1
Forage	1.5	1
Harold	5	1
Fierce Frank	7	1
Thomas Short-Tail	30	0.2
Brutal Bill	20	0.1

```
"parameters": [
  {
    "name": "abilities.16000.cooldown",
    "rangeMin": "0",
    "rangeMax": "2",
    "rangeAccuracy": "1",
    "minimise": "ignore",
    "weight": "1",
    "enabled": true
  }
],
```

Fig. 2: Sample of the specification language presenting a list with a single parameter for *ComPet*

game's actual code and is an accurate representation of combat in the game.

For the purposes of this paper, we designed a gauntlet that would have the player go through 9 beasts, each with varying skills and strengths (see Table I).

The main designer expectation is that after 3 of the gauntlet's beasts, the player would be unable to progress for a while, prompting them to gain any missing experience in player-versus-player (PVP) mode. There is another similar challenge with the 8th beast in the series. This experience could also be gained by fighting previously defeated beasts repeatedly. As a result, to simulate the PVP grind, our algorithm used this method of experience gathering. The beasts chosen for this experiment were the same as the first 9 beasts created for the actual game and would represent a player's first few hours playing the game.

The pet the player was given at the start of this gauntlet was level 1, with the default abilities a pet of that level would have in the actual game. The rate at which it gained attribute points was also predicated by the game's pace, so as to be realistic.

To get an idea of the current state of the campaign, we simulated an unchanged version of this gauntlet 100 times. The results are presented in Table II. The main metrics collected were the number of attempts required to defeat the beast in question and how many times, on average, did the pet manage to eventually beat the beast during a gauntlet run before the maximum number of battles was exceeded.

The designers looking at the results decided the difficulty levels of Ricktick and Thomas Short-Tail were too high and would need to be reduced, while keeping the rest of the beasts at current levels. The number of attempts it takes to defeat them and how often they are beaten every run would represent the metrics by which success is measured. All the metrics considered, alongside the desired values and their importance (Weight), are listed in Table III. The weights were arbitrarily chosen by the designers according to what they considered more important.

Elements of the game that were proposed for change were the health, endurance and ferocity of two beasts in the gauntlet (Ricktick and Thomas Short Tail's), as well as, for analysis purposes, the cooldown of one of Thomas Short-Tail's abilities (Vampiric Bite). This meant the evolution of 7 parameters: Ricktick's Health (RH), his Ferocity (RF), his Endurance (RE), Thomas Short Tail's Health (TH), his Ferocity (TF), his Endurance (RE), as well as the cooldown for Vampiric Bite (VC), all listed in Table IV.

To translate these requirements into something the algorithmic side could understand, we developed a specification language that would allow for a flexible representation of both parameters to change and metrics to measure success by. This made it easier for the developers and designers at MindArk to visualise and present both the search space and the desired results. Anecdotally, an aversion to command line tools with little to no documentation and a lack of time to learn new algorithms outside their product scope were mentioned.

C. Describing elements to be changed

A "parameter" is a variable in a game that has been deemed valuable for changing and balancing. An example of a list of parameters (with one element) can be seen in Figure 2.

For the designer a parameter's name is valuable, as it allows easy tracking and understanding of any changes proposed by the underlying algorithms. The name could be purely cosmetic, or it could be used to identify deeper elements in a game's design. For this paper, the name is used to describe the path to the appropriate JSON section in the *ComPet* results file.

From the point of view of an algorithm, what is being changed in the game is not important at all. The specification makes the assumption that the order in which parameters are defined is also the order they will be passed back to the game. What is important is knowing what values the parameters are allowed to have and their arithmetic precision.

An extra element available to the designer is the "enabled" flag. This is a simple "true" or "false" toggle that can be valuable when running multiple experiments. If the flag is set to "false", the parameter is not changed at all and is ignored by the algorithm.

Another option is the possibility of defining a parameter as not just a single value, but a list of values with a given length, all within the same ranges defined earlier. Should it be important, the values can be forced to be distinct, as this can

TABLE III: *ComPet* metrics to be used in evaluation alongside their desired values and weights

Metric	Name	Desired Value	Weight
M_1	Defeated Ricktick after # attempts	10	5
M_2	Defeated Forage after # attempts	1.5	1
M_3	Defeated Harold after # attempts	5	1
M_4	Defeated Fierce Frank after # attempts	7	1
M_5	Defeated Thomas Short-Tail after # attempts	15	5
M_6	Defeated Brutal bill after # attempts	20	1
M_7	# Wins per Run against Thomas Short-Tail	1	10

TABLE IV: *ComPet* parameters to be changed, their displacement ranges, their decimal accuracy and their weight in the fitness evaluation

Parameter	Variable	Min Range	Max Range	Accuracy	Weight
Vampiric Bite Cooldown	VC	+0	+2	10^0	0
Ricktick's Health	RH	-20	+20	10^0	0
Ricktick's Ferocity	RF	-10	+10	10^0	0
Ricktick's Endurance	RE	-10	+10	10^0	0
Thomas Short Tail's Health	TH	-30	+30	10^0	0
Thomas Short Tail's Ferocity	TF	-10	+10	10^0	0
Thomas Short Tail's Endurance	TE	-10	+10	10^0	0

be used to generate permutations. This option is available by setting the "listsize" property of the parameter.

Any number of these parameters can be the basis of a search for a balance trial, as this is how the algorithm communicates with the games.

When making changes to a game, a designer might be interested in making changes as small as possible. This is possible by setting the "minimise" flag to "minimise" and the weight option ($W_{\Delta i}$) to a non-zero value. This tells the underlying algorithm to consider the magnitude of parameter values when calculating a final score for the change set. Exposing this flag allows for experiments as those presented in the experiments on *Ms PacMan* and *StarCraft* by Morosan and Poli [3].

D. Evaluating success

Making changes to a game and playing the game with them is not enough. Once the game (or games, should one be not enough to assess success) has ended, the designer should have access to various values defining that play session. These could be anything from how often each player won, to how many times a weapon was used, to how long it took someone to solve a puzzle. We call these values "metrics". Metrics are how the game communicates with the algorithms. An example of a list of metrics (with one element) can be seen in Figure 3.

For this paper, similarly to parameters, a metric's name is used to describe the path to the appropriate JSON property in the *ComPet* results file.

To assess the success of a set of changes, a method of quantifying how close they are to optimal is needed. This is done by comparing the metrics generated after play to what the designer has deemed as optimal metrics. These comparisons are what the specification language defines as "evaluators". Each evaluator assesses a single metric, but a metric can be assessed by multiple evaluators. An example of an evaluator can be seen in Figure 4.

```
"metrics": [
  {
    "name": "DefaultPetBalanced.105.
   PetDefeatedBeastAfterAttemptAvg",
    "type": "Double"
  }
],
```

Fig. 3: Sample of the specification language presenting a list of metrics for *ComPet*

```
"evaluators": [
  {
    "name": "DefeatBeast105",
    "type": "AverageEvaluator",
    "metric": "DefaultPetBalanced.105.
   PetDefeatedBeastAfterAttemptAvg",
    "target": "10",
    "weight": "5",
    "enabled": true
  },
],
```

Fig. 4: Sample of the specification language presenting a list with a single evaluator for *ComPet*

Each evaluator has access to three values: a target optimal ($Target$) value, an optional extra parameter, used by some evaluators ($OptionalParam$), and a weight ($Weight$), used to define relative importance. They are passed a list of values ($Values$) that can be as small as a single element.

Similarly to parameters, evaluators can be marked as enabled or disabled.

The simplest evaluator is the "average" evaluator. It receives the list of values of a metric, gathered in one or multiple games, computes the average of those values, then compares the result to the given desired value. The resulting score is the absolute difference between the optimal value and the average of the metric, multiplied by the weight. Formally:

$$Eval_{Avg} = |Mean(Values) - Target| \times Weight$$

This evaluator can be used to compare a single value to a desired value. An example of such an evaluator could be that, after making changes to a pet, the designer wants it to end a certain fight, on average, with 20 health points when repeating that fight 20 times in a row.

Many other evaluators could be defined, such as one that calculates a median of one or more values, a standard deviation, or more complex calculations, such as the ratio of values in a list that are greater, or lesser, than a given threshold.

For this work, all evaluations are done using the intuitive average evaluator described previously.

After having calculated the result for each enabled evaluator, the final score is simply the sum of all the results. Similarly to other work in the area, lower values as better, with a total score of 0 representing a perfect solution.

The reason behind using a simple weighted sum instead of more complex multi-objective approaches, such as those utilised by Gravina and Loiacono [15], is due to simplicity. While the sum could allow for some objectives to dominate others and stop the underlying algorithm from exploring the search space more efficiently, it is a lot easier to explain and visualise. A straightforward system where someone untrained in machine learning optimisation can easily change the numbers and understand their impact is much more likely to be used outside of research.

E. Communicating with the games

While parameters and metrics are how the algorithms and games trade information from a logical point of view, a *bridge* is required to actually pass the digital data. This version of the framework currently defines a method that invokes an external command (such as running an executable or a script), as well as a distributed solution using message queues.

When invoking the external command, the game designer has access to several variables, such as: a comma separated list of the parameter values, a path to the JSON specification file and a random seed (a value that can be used to make it such that random number generators return the same sequence of numbers, valuable when attempting to replicate results). The external command in responsible of applying the relevant changes to the game's parameters, then launch the game (or games) and record the required metrics.

The message queue solution simply sets the "value" property of each enabled parameter, then sends the entire new JSON file to the message queue server. It is then distributed to a bridge script or application, as described in the previous paragraph, for playing. Once done, the bridge returns a JSON file with the appropriate metrics and their values. This assumes the bridge can read the JSON file and apply the appropriate changes. An example of the metrics a game could return can be seen in Figure 5.

```
{
  "metrics": {
    "DefaultPetBalanced.105.
    PetDefeatedBeastAfterAttemptAvg": "6",
    "DefaultPetBalanced.110.TotalPetWins
    ": "2"
  }
}
```

Fig. 5: Sample of JSON data sent by a game describing the requested metrics after completing play

MindArk Sweden graciously offered us access to *ComPet*'s battle simulator and default AI source code. Beyond a simple bridge to interact with their simulator, nothing much was required.

The bridge itself is an executable script that receives a list of changes from the algorithm and makes the required changes in the *ComPet* game, running the games, then reporting the results back to the algorithm, as described previously.

F. Genetic algorithm

We used the same approach to the genetic algorithm as Morosan and Poli [3], with a few minor adjustments, described next.

The evolutionary algorithm employed is a variant of a generational GA with two-point crossover [16] (applied with a rate of 40%), a specialised mutation operator (applied with a per-individual rate of 40%) and elitism (applied to the top 20% of the population).

The GA evolves a vector of displacements to the game's original values and they will be represented as the array (VC, RH, RF, RE, TH, TF, TE) and was constrained to only contain values between the different ranges specified in Table IV. Values outside of these ranges are not realistic for the tested scenarios.

The mutation operator was applied with a (per allele) mutation rate of 0.5 (meaning that on average 50% of the elements of an individual would be mutated). At each application of the operator, a displacement is randomly generated by a random number generator within the range of acceptable values for that allele and replaces the corresponding parameter value.

Experiments used tournament selection, with a tournament size of 6. The population size was 50, with no more than 50 generations for each run.

G. Fitness evaluation

As described in previous sections, for each evaluation, the parameters in an individual were decoded and games were

played on a corresponding version of the game. This was done by passing those parameters to the bridge application and waiting for the metrics to be returned, then running the evaluators on them.

As opposed to work done by Morosan and Poli on *Ms Pac-Man* and *StarCraft*, favouring small changes to existing parameters was not required in the *ComPet* experiment. The most important aspect is having the final metrics achieve the designer-set values. As a result, for each of the metrics, the closer they are to the desired values, the better the fitness is.

Using the average evaluator described in section II-D on each resulting metric, then summing the evaluations together, we have a fitness score assigned to the parameter set.

Formally, the fitness function for *ComPet* can be written as:

$$Fitness_{ComPet} = \sum_{i=1}^{m} |M_i - DM_i| \times W_i \qquad (4)$$

where m = number of metrics considered for comparison (7 for this experiment), M_i = value of metric i from the games played with the evolved version of the game, DM_i = desired value for the metric i, and W_i = weight or importance given to that metric.

The GA considers smaller fitness values to be better, with 0 representing a perfect solution.

III. Results

The main experiment involved balancing a section of the *ComPet* campaign to fit the designer's requirements. The goal, following the requirements and discussion presented in Section II-B, was to find good changes to the proposed parameters, in respect to the metrics presented in Table III.

In this experiment 10 runs were done, each with a different seed. This took a lot of computational effort, as the simulations took a significant time to complete each game.

Looking at the changes done by the best individual in each run (Table V) and the individual components of the multi-objective fitness function (Table VI), the immediate observation resulting from this experiment is that changing the beast Ricktick can be done in various ways.

Most runs did not, on average, require big changes to Ricktick's statistics, suggesting that that particular beast is not too far from a balanced state and requires only small adjustments. Indeed, the few runs where Ricktick received very big changes resulted in worse fitnesses for that particular objective (M_1).

The interesting results come when analysing the suggestions for beast Thomas Short-Tail. The runs were evenly split between two major strategies: to weaken the beast by making its main attack usable less often, but increase its other statistics (mostly health), or the exact opposite, where no change is made to the cooldown of that ability, but penalties given to health, ferocity and/or endurance.

These two strategies are also obvious when visualising the 10 runs as a graph. By using Euclidean distance to find the similarity between runs, then an algorithm such as GMap [17]

TABLE V: Main *ComPet* experiment results, highlighting the changes recommended by each run. Green highlighting represents big changes by adding to the original value, red highlighting represents big changes by subtracting from the original value, while colours in-between represent smaller intensity changes

Run	VC	RH	RF	RE	TH	TF	TE
1	+1	−2	+3	+3	+16	−9	−5
2	+1	+5	+3	−7	+16	+6	−5
3	+1	+5	+10	+2	+10	+10	+10
4	+1	−7	+10	+0	+30	+10	−6
5	+0	+2	+6	−1	−11	+5	−6
6	**+0**	**+2**	**+5**	**−1**	**−14**	**−2**	**+8**
7	+0	+2	−4	+7	−15	+5	−5
8	+0	−11	+10	+7	−11	−1	−5
9	+0	+3	−10	+0	−14	−6	−4
10	+1	−7	+10	−10	+7	+10	−4

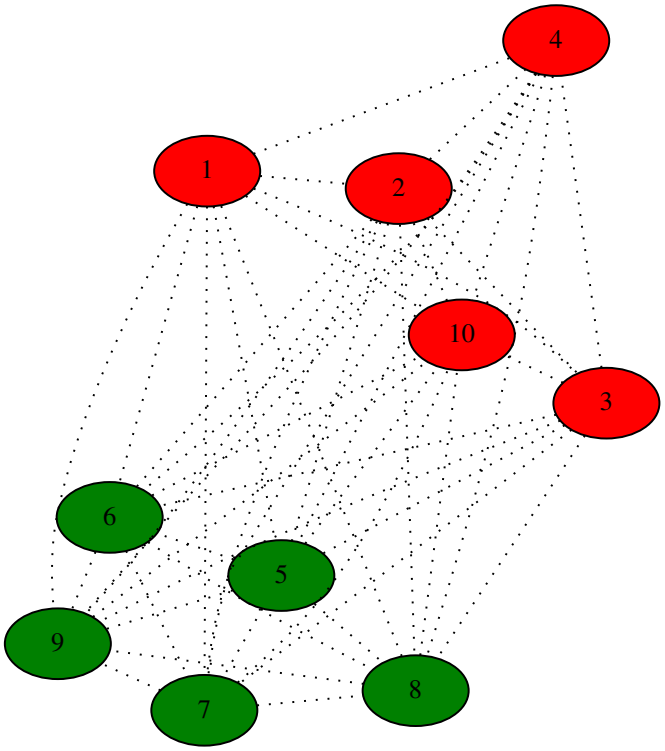

Fig. 6: Similarity map representing the Euclidean distance between the 10 *ComPet* run parameter change suggestions, with colour-coded clusters highlighting the 2 main balancing strategies proposed

to display the run results as a graph, a similarity map can be generated, such as the one in Figure 6.

Such a visual tool can further allow a designer to analyse multiple run results and observe patterns in the suggestions.

For this set of runs, one of the immediate conclusions was that changing the cooldown of Vampiric Bite (parameter VC) by increasing it would not result in desirable results. This parameter could now be omitted in future runs.

It is also worth noting how some fitness objectives, particularly M_1 (number of attempts required to defeat Ricktick)

TABLE VI: Main *ComPet* experiment results, highlighting the individual fitness objectives and scores achieved by each run

Run	M_1	M_2	M_3	M_4	M_5	M_6	M_7	Fitness
1	2	0.5	1	3.6	1.25	3.33	6	17.68
2	1.25	0	0.25	0.5	1.25	12.66	6	21.91
3	6.25	0.5	0.5	2	0	7.25	6	22.5
4	4	0.25	0.5	2.5	5	3.5	6	21.75
5	3	0.3	0.6	3	3	5.75	5	20.65
6	**2**	**0.75**	**0.5**	**1**	**3.75**	**1.33**	**6**	**15.33**
7	5	0.3	3.4	2.4	1	2	5	19.1
8	5	5.3	0.6	2	0	1	5	18.9
9	2	0.7	1	1.2	3	5.75	5	18.65
10	0	0.1	0.8	2.4	1	12.6	5	21.9

and M_6 (number of attempts required to defeat Brutal Bill), were much harder to optimise compared to the rest (see Table VI). The second one is most likely due to the fact that by making the previous fight simpler, it allowed a weaker pet to arrive at the Brutal Bill fight, increasing the number of attempts required before success. These would require additional analysis from the designer, as different experiments or parameter sets might be necessary to achieve the goals.

Metric M_7 (number of wins on average against Thomas Short Tail) had the most consistent results, however nowhere near optimal. This points towards more changes being required before the goal of getting a win every run is achievable with the current parameter constraints.

Every other metric was much easier to maintain at desired values.

These are all results that the designer will find useful, as they offer valuable insight into the relationships between the various parameters. The designers behind *ComPet* decided to apply the results from run 6 with slight tweaks of their own, as they were closest to what they were hoping to achieve. They also planned on designing more experiments in the future, while also providing valuable feedback on how these algorithms could be even better for commercial use. This included a clear call for simpler tools and better explained pipelines for receiving balance suggestions, mentioning that the specification language was a step in the right direction and that going further would be a win for both academia and industry.

The algorithms themselves, while important, were not their highest priority. Most valuable were the suggestions offered to them and how well explained they were. The presence of quantifiable metrics and easy visualisation was beneficial to understanding the algorithm's suggestions.

IV. Conclusion and future work

By integrating a specification language and a bridge with an existing genetic algorithm developed in academia, we were able to demonstrate applicability in industry through a cooperation with MindArk and their commercial product, *ComPet*. The successful application of these techniques to a commercial game is reassuring, as the goal is for them to be valuable in the real world, helping the development of games. A lot of feedback was received from MindArk and much of it has already been implemented by us to great results.

The specification language presented may seem simplistic, but it is critical to remember that simplicity will allow for mainstream adoption of academic research in industry and potential for further growth. Tighter cooperation between industry and academia can only benefit both.

While each run, regardless of strategy evolved, presented the best result as a list of numeric changes, there are potentially better ways of presenting this information. Replacing the numbers with slightly broader suggestions (such as replacing a $+16$ to Thomas Short-Tail's Health with "Significantly increase Thomas Short-Tail's Health") can make results a lot easier to interpret by designers. This will be tested in future research.

For performance, the bottleneck was in no way related to the length of the arrays, but in playing out the games themselves for the fitness evaluation. Better hardware or access to a game's source code would also greatly increase the speed of an evaluation, allowing for more individuals in a population, more games to be played, or more generations to be run. A single individual in this experiment needed about 15 seconds to have its fitness evaluated.

The area of automated game balancing is being explored in depth at this point in time by many researchers and this can only benefit the games development world. Manual testing will, almost certainly, never be obsolete, but designers will be able to focus on much more interesting tasks while letting computational intelligence do the less exciting elements of balance.

This successful use of research in an industrial context is a sign that not only is there need for more research on the topic, but that also there can be demand for it in the real world, given proper presentation and tools.

References

[1] G. Mountain, "Tactics in Fable Legends," 2015. [Online]. Available: http://gwaredd.github.io/nuclai_mcts/root/index.html

[2] C. Kerr, "DeepMind wants to answer the big ethical questions posed by AI," 2017. [Online]. Available: https://www.gamasutra.com/view/news/307008/DeepMind_wants_to_answer_the_big_ethical_questions_posed_by_AI.php

[3] M. Morosan and R. Poli, "Automated Game Balancing in Ms PacMan and StarCraft Using Evolutionary Algorithms." Springer, Cham, 2017, pp. 377–392. [Online]. Available: http://link.springer.com/10.1007/978-3-319-55849-3_25

[4] I. Schreiber, "Game Balance Concepts," 2010. [Online]. Available: https://gamebalanceconcepts.wordpress.com/2010/07/07/level-1-intro-to-game-balance/

[5] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation: A taxonomy and survey," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 172–186, 2011.

[6] M. Cook, "A Vision For Continuous Automated Game Design," 7 2017. [Online]. Available: http://arxiv.org/abs/1707.09661

[7] T. Mahlmann, J. Togelius, and G. N. Yannakakis, "Evolving Card Sets Towards Balancing Dominion," *2012 IEEE Congress on Evolutionary Computation*, pp. 1 – 8, 2012.

[8] V. Volz, G. Rudolph, and B. Naujoks, "Demonstrating the Feasibility of Automatic Game Balancing," 3 2016. [Online]. Available: http://arxiv.org/abs/1603.03795

[9] M. Beyer, A. Agureikin, A. Anokhin, C. Laenger, F. Nolte, J. Winterberg, M. Renka, M. Rieger, N. Pflanzl, M. Preuss, and V. Volz, "An Integrated Process for Game Balancing," *IEEE Conference on Computational Intelligence and Games*, 2016.

[10] M. Morosan and R. Poli, "Evolving a designer-balanced neural network for Ms PacMan," in *2017 9th Computer Science and Electronic Engineering (CEEC)*, 9 2017, pp. 100–105.

[11] H. Chen, Y. Mori, and I. Matsuba, "Solving the Balance Problem of On-Line Role-Playing Games Using Evolutionary Algorithms," *Journal of Software Engineering and Applications*, vol. 05, no. 08, pp. 574–582, 2012. [Online]. Available: http://www.scirp.org/journal/PaperDownload.aspx?DOI=10.4236/jsea.2012.58066

[12] "Metaheuristics in the Large," 2018. [Online]. Available: http://www.mitlware.org/

[13] J. Swan, S. Adriaensen, M. Bishr, E. K. Burke, J. A. Clark, P. De Causmaecker, J. Durillo, K. Hammond, E. Hart, C. G. Johnson, Z. A. Kocsis, B. Kovitz, K. Krawiec, S. Martin, J. J. Merelo, L. L. Minku, E. Ozcan, G. L. Pappa, E. Pesch, P. García-Sánchez, A. Schaerf, K. Sim, J. E. Smith, T. Stützle, S. Voß, S. Wagner, and X. Yao, "A Research Agenda for Metaheuristic Standardization," in *MIC 2015: The XI Metaheuristics International Conference*, 2015. [Online]. Available: http://www.cs.nott.ac.uk/~pszeo/docs/publications/research-agenda-metaheuristic.pdf

[14] MindArk, "Compet Game," 2018. [Online]. Available: https://competgame.com/

[15] D. Gravina and D. Loiacono, "Procedural weapons generation for unreal tournament III," in *2015 IEEE Games Entertainment Media Conference (GEM)*. IEEE, 10 2015, pp. 1–8. [Online]. Available: http://ieeexplore.ieee.org/document/7377225/

[16] D. E. Goldberg and others, *Genetic algorithms in search optimization and machine learning*. Addison-wesley Reading Menlo Park, 1989, vol. 412.

[17] E. R. Gansner, Y. Hu, and S. Kobourov, "GMap: Visualizing graphs and clusters as maps," in *2010 IEEE Pacific Visualization Symposium (PacificVis)*. IEEE, 3 2010, pp. 201–208. [Online]. Available: http://ieeexplore.ieee.org/document/5429590/