Metagame Autobalancing for Competitive Multiplayer Games

Daniel Hernandez*, Charles Takashi Toyin Gbadamosi†, James Goodman‡
and James Alfred Walker*, *Senior Member, IEEE*

*Department of Computer Science, University of York, UK. {dh1135, james.walker}@york.ac.uk
†Department of Computer Science, Queen Mary University of London, UK. {c.t.t.gbadamosi, james.goodman}@qmul.ac.uk

***Abstract*—Automated game balancing has often focused on single-agent scenarios. In this paper we present a tool for balancing multi-player games during game design. Our approach requires a designer to construct an intuitive graphical representation of their meta-game target, representing the relative scores that high-level strategies (or decks, or character types) should experience. This permits more sophisticated balance targets to be defined beyond a simple requirement of equal win chances. We then find a parameterization of the game that meets this target using simulation-based optimization to minimize the distance to the target graph. We show the capabilities of this tool on examples inheriting from Rock-Paper-Scissors, and on a more complex asymmetric fighting game.**

I. Introduction

Achieving game balance is a primary concern of the game designer, but balancing games is a largely manual process of trial and error. This is especially problematic in asymmetric multiplayer games where perceived fairness has a drastic impact on the player experience. Changes to individual game elements or rules can have an impact on the balance between high-level strategies that depend on these, but this impact is unknown before changes are made and can only be guessed at by designers through experience and intuition. We term this balance between emergent high-level strategies the 'Meta-game balance'. While in-house tools can be built for the adjustment and authoring of individual game elements. There are no tools for balancing and adjusting -game elements.

Game balancing takes a lot of time and resources, with current trends indicating a systematic increase in the cost of game development [1]. It is reliant on human intuition and expert knowledge to estimate how changes in the game mechanics affect emergent gameplay. Human play testing as part of this process is time consuming, requiring many human testers for long play-sessions, which grow longer with more complex games. In short, human play testing does not scale.

An alternative approach to the discovery of meta-game changes that arise from game changes is through data analytics. Large scale multiplayer titles that have access to large quantities of player data can use a variety of techniques to make judgements about the state of the meta-game and provide designers with insight into future adjustments, such as [2].

Thank you, Ozan Vardal, Nick Ballou and Sebastian Berns for your generous help in coding Workshop Warfare. This work was funded by the EPSRC Centre for Doctoral Training in Intelligent Games and Game Intelligence (IGGI) EP/L015846/1 and Digital Creativity Labs.

978-1-7281-4533-4/20/$31.00 ©2020 Crown

There are, however, several problems with this approach. Analytics can only discover balance issues in content that is live, and by that point balance issues may have already negatively impacted the player experience: this is a reactive approach and not a preventive one. Worse, games which do not have access to large volumes of player data - less popular games - cannot use this technique at all.

Furthermore, the process of data analytics itself is not typically within the skill-set of game designers. It is common for studios that run multiplayer games to hire data scientists to fill this need. This, in combination with the trial and error nature of the balance process, results in increased costs, becoming as a bottleneck for the development of new content.

The importance of meta-game balance and the aforementioned issues motivate alternate approaches to game balance. This paper presents one such alternative - an automated simulation-based approach to meta-game balance of multiplayer games. Our approach allows designers to directly specify a meta-game balance state and have the game parameters that would create the desired meta-game be discovered automatically by a group of agents.

II. Preliminary notation

Cursive lowercase letters represent scalars (n). Bold lowercase, vectors ($\boldsymbol{\pi} \in \mathbb{R}^n$). Bold uppercase, matrices ($\boldsymbol{A} \in \mathbb{R}^{n \times n}$).

A. Game parameterization

Every video game presents a (potentially very large) number of values that characterize the game experience, which we shall refer to as **game parameters**. These values can be numerical (such as gravitational strength, movement speed, health) or categorical (whether friendly fire is activated, to which team a character belongs). As a designer, choosing a good set of parameters can be the difference between an excellent game and an unplayable one. We let $E_{\boldsymbol{\theta}}$ denote a game environment, parameterized by an n-sized parameter vector $\boldsymbol{\theta} \in \{\Pi_{i \leq n} \Theta_i\}$, where $\{\Pi_{i \leq n} \Theta_i\}$ represents the joint parameter space, and Θ_i the individual space of possible values for the ith parameter in $\boldsymbol{\theta}$.

B. Meta-games

What a **meta-game** is can mean different things to different players. For example in deck-building games such as Hearthstone, the 'meta' is usually interpreted to indicate which decks are currently popular or especially strong; while in EVE Online an important part of the 'meta' is player diplomatic alliances, as well as which ship types are good against which others. See [3] for a good discussion of this notation.

In this work we define a meta-game as a set of high-level strategies that are abstracted from the atomic game actions. Reasoning about a game involves thinking about how each individual action will affect the outcome of the game. In contrast, a meta-game considers more general terms, such as how an aggressive strategy will fare against a defensive one. In meta-games, high level strategies are considered instead of primitive game actions. Take a card game like Poker. Reasoning about a Poker meta-game can mean reasoning about how bluff oriented strategies will deal against risk adverse strategies.

The level of abstraction represented in a meta-game is defined by the meta-game designer, and the same game can allow for a multitude of different levels of abstraction. For instance, in the digital card game of Hearthstone, meta-strategies may correspond to playing different deck types, or whether to play more offensively or defensively within the same deck. A game designer may want to ensure that no one deck type dominates, but be happy that a particular deck can only win if played offensively.

C. Empirical win-rate matrix meta-games

An interesting meta-game definition that has recently received attention in multiagent system analysis [4] defines a normal form game over a population of agents $\boldsymbol{\pi}$, such that the action set of each player corresponds to choosing an agent $\pi_i \in \boldsymbol{\pi}$ from the population to play the game for them. How these agents were created is not relevant to us; these agents could use hand-crafted heuristics, be trained with reinforcement learning, evolutionary algorithms or any other method.

Let $\boldsymbol{W_\pi} \in \mathbb{R}^{n \times n}$ denote an **empirical win-rate matrix**. The entry $w_{i,j}$ for $i, j \in \{n\}$ represents the win-rate of many head-to-head matches of policy π_i when playing against policy π_j for a given game. An empirical win-rate matrix $\boldsymbol{W_\pi}$ for a given population $\boldsymbol{\pi}$ can be considered as a payoff matrix for a 2-player zero-sum game. An empirical win-rate matrix can be defined over two (or more) populations $\boldsymbol{W_{\pi_1,\pi_2}}$, such that each player chooses agents from a different population. We can investigate the strengths and weaknesses of each these agents in this kind of meta-game using game-theoretical analysis.

An **evaluation matrix** [5] is a generalization of an empirical win-rate matrix. Instead of representing the win/loss ratio between strategies, it captures the payoff or score obtained by both the winning and losing strategy. That is, instead of containing win-rates for a given set of agents, an entry in an evaluation matrix $a_{ij} \in \boldsymbol{A}$ can represent the score obtained by the players.

D. Empirical Response Graphs

A **directed weighted graph** of $v \in \mathbb{N}^+$ nodes can be denoted by an adjacency matrix $\boldsymbol{G} \in \mathbb{R}^{v \times v}$. Each row i in $\boldsymbol{G}$ signifies the weight of all of the directed edges stemming from node i. Thus, $g_{i,j} \in \mathbb{R}^+$ corresponds to the weight of the edge connecting node i to node j, where $1 \leq i, j \leq v$.

Given an evaluation matrix $\boldsymbol{A_\pi}$ computed from a set of strategies (or agents) $\boldsymbol{\pi}$, let its **response graph** [6] represent the dynamics [4] between agents in $\boldsymbol{\pi}$. That is, a representation of which strategies (or agents) perform favourably against which other strategies in $\boldsymbol{\pi}$. In a competitive scenario, a response graph shows which strategies win against which others. As a graph, each strategy i is represented by a node. An edge connecting node i to node j indicates that i dominates j. The weight of the edge is a quantitative metric of how favourably strategy i performs against j. Figure 1a shows a response graph for the game of Rock-Paper-Scissors (RPS).

A response graph can be readily computed from an evaluation matrix. Each row i in an evaluation matrix $\boldsymbol{A}$ denotes which strategies i both wins and loses against, the former being indicated by positive entries and the latter by negative ones. Therefore, generating a response graph $\boldsymbol{G}$ from an evaluation matrix $\boldsymbol{A}$ is as simple as setting all negative entries of $\boldsymbol{A}$ to 0 such that, for instance, $\boldsymbol{A} = \left(\begin{smallmatrix} 1 & -2 \\ 2 & -1 \end{smallmatrix}\right)$, becomes $\boldsymbol{G} = \left(\begin{smallmatrix} 1 & 0 \\ 2 & 0 \end{smallmatrix}\right)$.

E. Graph distance

There is a rich literature on measuring distance between graphs [7]. We concern ourselves here with a basic case. We are interested in measuring the distance between two graphs which share the same number of nodes, $\boldsymbol{G_1}, \boldsymbol{G_2} \in \mathbb{R}^{v \times v}$, and differ *only* in the weight of the edges connecting nodes. Because graphs can be represented as matrices, we look at differences between matrices. We denote the distance between two graphs $\boldsymbol{G^1}$ and $\boldsymbol{G^2}$ by $d(\boldsymbol{G^1}, \boldsymbol{G^2}) \in \mathbb{R}$. Equation (1) represents the average absolute edge difference and (2) represents the mean squared difference (MSE).

$$\frac{\sum_{i,j} |g^1_{ij} - g^2_{ij}|}{n} \quad (1) \qquad\qquad \frac{\sum_{i,j} (g^1_{ij} - g^2_{ij})^2}{n} \quad (2)$$

Preliminary results showed no empirical difference between distance metrics (1) and (2). Thus, we report only the results where MSE (Equation 2) was used.

III. Autobalancing

In this section we present our autobalancing algorithm in its most general form.

A. Optimization setup

Let $E_{\boldsymbol{\theta}}$ be a game environment parameterized by vector $\boldsymbol{\theta} \in \mathbb{R}^n$, whose possible values are bound by vectors $\boldsymbol{\theta}^{min}$ and $\boldsymbol{\theta}^{max}$. Let $\boldsymbol{G_t}$ denote the target metagame response graph presented by a game designer for game $E_{\boldsymbol{\theta}}$. Let $\boldsymbol{G_\theta}$ represent the empirical metagame response graph produced from a set of gameplaying agents $\boldsymbol{\pi}$ for game $E_{\boldsymbol{\theta}}$, where each agent corresponds to a node in the graph $\boldsymbol{G_\theta}$. Finally, let $\mathcal{L}(\cdot,\cdot)$ represent a cost or distance function between two graphs.

The mathematical formulation for finding a parameter vector $\boldsymbol{\theta}$ which yields a metagame for a game environment $E_{\boldsymbol{\theta}}$ respecting designer choice $\boldsymbol{G_t}$ is a constrained non-linear optimization problem:

$$\arg\min_{\boldsymbol{\theta}} \quad \mathcal{L}(\boldsymbol{G_\theta}, \boldsymbol{G_t}) \tag{3}$$

$$s.t \quad \theta_i^{min} \leq \theta_i \leq \theta_i^{max}\ \forall i \in \{|\boldsymbol{\theta}|\} \tag{4}$$

Algorithm 1: Automated balancing algorithm.

Input: *Target designer meta-game response graph*: $\boldsymbol{G_t}$
Input: *Ranges for each parameter*: $\boldsymbol{\theta}^{min}, \boldsymbol{\theta}^{max}$
Input: *Convergence threshold*: ϵ

```
1  Initialize game parameterization θ_0;
2  Initialize best estimate θ_best, L_best = θ_0, ∞;
3  Initialize observed datapoints D = [ ];
4  repeat
5      Train agents π inside E_{θ_t}, for each node in G_t;
6      Construct evaluation matrix A_{θ_t} from π;
7      Generate response graph G_{θ_t};
8      Compute graph distance d_t = L(G_{θ_t}, G_t);
9      Add new datapoint D = D ∪ (θ_t, d_t);
10     if d_t < L_best then
11         Update best estimate θ_best, L_best = θ_t, d_t;
12     end
13     θ_{t+1} = update(θ_t, D);
14 until L(G_{θ_t}, G_t) < ε;
15 return θ_best;
```

There are four notes to be made about our algorithm:

1) **It can be parallelized**: multiple parameter vectors can be evaluated simultaneously.
2) **It allows for initial designer choice**: such that designers can designate an initial parameter vector and a prior over the search space, which can lead to speedup in the convergence of the algorithm.
3) **An arbitrary subset of the game parameters can be fixed**: $\boldsymbol{\theta}$ can represent a subset of the entire game parameters. This is important if there are certain core aspects of a game that the designer does not want to be altered throughout the automated game balancing.
4) **Deterministic results are not guaranteed**. There are three potential sources of stochasticity, the game dynamics $E_{\boldsymbol{\theta}}$, the agent policies $\boldsymbol{\pi}$ and the optimizer's parameter choices (line 13 of Algorithm 1).

There are two potential bottlenecks in Algorithm 1 in terms of the computational requirements of (1) the construction of the evaluation matrix and (2) the update of the parameter vector. The main computational burden in (1) comes from the fact that computing each entry in an evaluation matrix $a_{ij} \in \boldsymbol{A}$ require running many game episodes played by agents i and j, with the cost of computing $\boldsymbol{A}$ growing exponentially with respect to the number of agents.

B. Choosing an optimizer

We want to emphasize that our algorithm can use any black-box optimization method. To compute updates to our parameter vector $\boldsymbol{\theta}$ we use Bayesian optimization. Specifically, we use the algorithm Tree-structured Parzen Estimator [8], as implemented in the Python framework Optuna [9], but this could be replaced with any other optimization method. Most commonly in the literature of automated game balancing, evolutionary algorithms have been used [10].

C. Choosing a metagame abstraction

For most games, there are many possible abstractions (and levels of abstraction) available when deciding what the metagame captured by the target response graph represents.

Choosing the abstraction may not be obvious, but we argue that reasoning about metagames is a necessary task in balancing any multi-agent game. On a positive note, the fact that metagames can be represented at many levels of abstraction grants our method the versatility to generalize to various stages of balancing. That is to say, our method can be used at different points of game development to balance different aspects of the game.

Generally each node on the response graph represents a specific strategy, unit or game-style. A possible target response graph could symbolize the interactions between players or agents trained to represent different in-game "personas" [11], where a "persona" representing a different reward scheme for an agent. In an RPG each node of the response graph might represent a character class; Paladin, Wizard, Sniper etc., as we seek to balance these against each other. At a lower level, each node might represent an individual weapon.

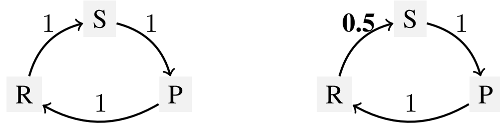
During auto-balancing we train an AI to play each of the strategies/units that the nodes represent as well as possible, where this will often mean 'winning', but could use some other balance target such as 'gold gained', or 'length of fight'.

D. Generating game-playing agents

As specified in Section II, in order to compute an evaluation matrix $\boldsymbol{A}$ for a given game $E_{\boldsymbol{\theta}}$ we require a set of gameplaying agents $\boldsymbol{\pi}$. These could be hand-crafted heuristic agents, or agents trained via reinforcement learning or evolutionary algorithms [12], [13]. The algorithmic choice for how to train these agents is orthogonal to the usage of our method. However, we acknowledge that the creation of these agents can be a significant engineering and technical effort.

IV. Motivating examples

In this section we present basic examples of our automated balancing algorithm. For simplicity, we assume that all parameters in the following examples are bound between $[-1, +1]$. As a graph distance metric we use $\mathcal{L}(\cdot,\cdot) = MSE(\cdot,\cdot)$ from Equation 2.



(a) RPS (b) Biased RPS

Fig. 1: Target graphs for the 2 motivational examples

1) Rock Paper Scissors: Imagine we want to create the game of Rock Paper Scissors [1]. As a designer choice, we want paper to beat rock, rock to beat scissors and scissors to beat paper, with mirror actions negating each other. Such strategic balancing is captured in Figure 1a. These interactions can be represented as a 2-player, symmetric, zero-sum normal form game $\boldsymbol{E}^{RPS}_{\boldsymbol{\theta}}$, parameterized by $\boldsymbol{\theta} = [\theta_{rp}, \theta_{rs}, \theta_{ps}]$. Where θ_{rp} denotes the payoff for player 1 when playing Rock against Paper, θ_{rs} when playing Rock against Scissors and θ_{ps} when playing Paper against Scissors. The normal form parameterized version of RPS is captured in Equation 5. We ask the question: Which parameter vector $\boldsymbol{\theta}$ would yield a game $\boldsymbol{E}^{RPS}_{\boldsymbol{\theta}}$ balanced as in Figure 1a?.

$$\boldsymbol{E}^{RPS}_{\boldsymbol{\theta}} = \begin{bmatrix} \cdot & R & P & S \\ R & 0 & \theta_{rp} & \theta_{rs} \\ P & -\theta_{rp} & 0 & \theta_{ps} \\ S & -\theta_{rs} & -\theta_{ps} & 0 \end{bmatrix} \tag{5}$$

We begin by assuming the target balance response graph $\boldsymbol{G_t}$ from Figure 1a is given by a game designer. Lacking any informed priors, we start by sampling a random valid parameter vector, say, $\boldsymbol{\theta_0} = [-1, 1, 0]$. We then generate an evaluation matrix by pitting Rock, Paper and Scissors against each other, yielding $\boldsymbol{A_{\theta_0}} = \begin{pmatrix} 0 & -1 & 1 \\ 1 & 0 & 0 \\ -1 & 0 & 0 \end{pmatrix}$, whose response graph $\boldsymbol{G_{\theta_0}} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$ is depicted in Figure 2. We proceed by computing the distance between $\boldsymbol{G_{\theta_0}}$ and $\boldsymbol{G_t}$, $d_{\boldsymbol{\theta_0}} = \mathcal{L}(\boldsymbol{G_{\theta_t}}, \boldsymbol{G_t}) = 0.25$. Using this new datapoint ($\boldsymbol{\theta_0}$, $d_{\boldsymbol{\theta_0}}$) we update our black box optimization model, which in our case is Bayesian optimization, and sample a new $\boldsymbol{\theta_1}$. This process is looped until convergence or an arbitrary computational budget is spent.

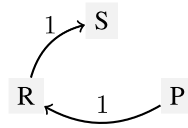


Fig. 2

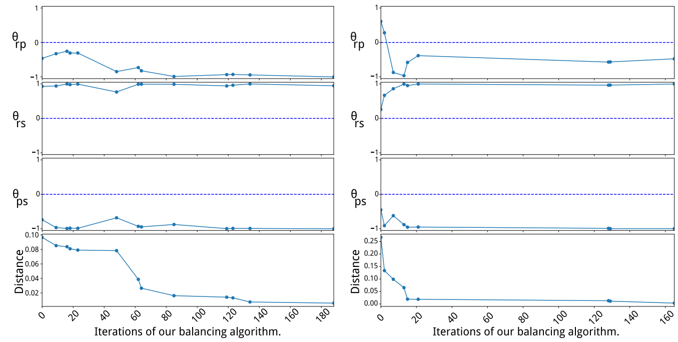
2) Biased Rock Paper Scissors: Consider another version of Rock Paper Scissors where we want to *weaken* the strength of playing Rock, as denoted in Figure 1b. For our algorithm, this amounts to discovering a lower payoff θ_{rp} obtained by playing Rock against Scissors.

Figure 3 shows the progression of parameter values $\boldsymbol{\theta}$ computed for problems 1) RPS and 2) Biased RPS, described above. With 1% tolerance, our method converges to the correct parameter values within 180, and 160 respectively.

V. Usage on a real game

The parameters optimized in the previous section directly influenced the payoff obtained by the agents playing the game.

[1]https://en.wikipedia.org/wiki/Rock%E2%80%93paper%E2%80%93scissors



(a) RPS (b) Biased RPS

Fig. 3: Progression of balance parameters and distance to target graph. Only parameters which improved with respect to the previous best estimate are plotted. The target parameter values for (a) and (b) respectively are: $[-1, 1, -1]$, $[0.5, 1, -1]$.

This is not a realistic scenario. The game parameters that designers can directly change impact game mechanics, which only indirectly affect the outcome of a game. Therefore, for the remainder of this section, we don our game designer hat, to showcase a usage of our algorithm in a realistic challenge.

A. *Workshop Warfare: a more realistic game*

Workshop Warfare[2] is a 2-player, zero-sum, symmetric, turn based, simultaneous action game. The theme of the game is a 1v1 battle between robots on a 5x5 grid with the objective of depleting the opponent's health[3]. Each player chooses 1 out of 3 available robots (Figure 4) to fight the opponent's robot of choice, with each robot featuring a different style of play. All robots feature the same action space: standing still (S), moving up (U), down (D), left (L), right (R) and a special action (A). The special action (A) varies per robot and will be explained later.

Workshop Warfare works on a "tick" basis. Each bot has an associated number of ticks shared across all action, representing how many in-game ticks must elapse between actions. This property can be thought as a time cost or robot "speed". A bot is said to be "sleeping" during the period that it cannot take actions. Standing still (S) has no cost, meaning that it allows the bot which took that action to take another action on the next tick. This allows for a degree of strategic depth.

To clarify the tick based system, imagine an scenario with two bots, with 2 and 4 ticks respectively. They both select a (U) action, moving upwards by 1 square in the grid. The next tick will elapse without anything happening, as both bots are "sleeping". On the next tick, bot 1 will be able to act again, followed by another tick with both bots sleeping. On the next tick both bots will be able to act again.

[2]The game is open source, and follows an OpenAI Gym interface [14]: https://github.com/Danielhp95/GGJ-2020-cool-game

[3]Akin to TV shows like Battle Bots https://www.wikiwand.com/en/BattleBots

There are no time restrictions placed upon the players at the time of selecting an action. This makes it amenable for forward planning methods that use a given computational budget to decide on what action to take. Thus, when autobalancing, this budget can be scaled without affecting the flow of the game, this is further explained in Section VI-C.

We now describe all three bots, whose in-game sprites are shown in Figure 4. **Torch bot** is equipped with a damaging blow torch, and can shoot a continuous beam of fire of limited range in all four directions for a short amount of time. **Nail bot** has a nail gun, and can shoot nails in all four directions at once. When fired, each nail travels in a fixed direction, at a speed of one grid cell per tick, independently of the bot's later movement and deals damage if they hit the opponent. **Saw bot**'s spikes deal damage by being adjacent to the opponent. Its ability is to temporarily increase its damage.

B. *Game parameterization*

All bots share some common parameters, although their individual values can differ from bot to bot. Other parameters are bot specific and relate to a bot's special action (A).

Common parameters

- *Health*: Damage a bot can sustain before being destroyed.
- *Cooldown*: After the special action (A) is activated, number of ticks that need to elapse before that action can be used again.
- *Damage*: Damage dealt by flames (Torch bot), nails (Nail bot) or spikes (Saw bot).
- *Ticks between moves*: Number of ticks that need to elapse before another action can be taken.

Bot-specific parameters

- *Torch range*: Length of the blow torch flame, in number of grid squares. (Torch bot)
- *Torch duration*: number of ticks the torch flame is active (Torch bot).
- *Damage buff*: Temporary change in damage dealt (Saw bot).
- *Duration*: Duration of buff to damage (Saw bot).

The parameters that were optimized in Section IV were real valued ($\mathbb{R}$), whereas all the parameters in this section are natural numbers ($\mathbb{N}$). The number of parameter combinations inside the parameter space would be prohibitively time consuming for any human designers to manually explore. We now apply our autobalancing method to automate this process.

(a) Nail Bot (b) Saw Bot (c) Torch Bot

Fig. 4: Eligible characters in Workshop Warfare.

VI. Experiments on Real Game

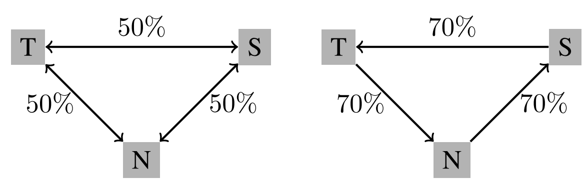
We first choose our level of abstraction, and what elements we want to balance as game designers. For these experiments, we choose to balance the win-rates between all bot matchups. We want these win-rates to represent the win-rate between rational competitive players, that is, players who play to win understanding that their opponent also aims for the same goal. As a proxy of rational players we use AI agents controlled by Monte Carlo Tree Search (MCTS), as detailed below.

(a) Torch bot VS Saw bot. (b) Torch bot vs Nail bot.

Fig. 5: Screenshots of the game.

This level of abstraction differs from the motivating examples from Section IV. In Section IV we directly modified the deterministic payoff obtained by any matchup (i.e, Rock vs Paper, etc). In these examples we instead aim to balance the win-rate obtained when each bot type is matched up against each of the other bots. This win-rate is emergent from the precise parameter settings that we can control, listed in Table II. This is a much more realistic design scenario.

A. *Target meta-game balance*



(a) Fair balance (b) Cyclic behaviour

Fig. 6: Target graphs for the 2 experiments on Workshop Warfare. Note the bi-directionality of graph 6a

We run two experiments, each corresponding to a different design goal. We will attempt to find the parameter vector $\boldsymbol{\theta}$ which yields a meta-game balance, in terms of bot win-rates, as described by the response graphs in Figure 6. Each element $\theta \in \boldsymbol{\theta}$ corresponds to a game parameter in Table II.

The two design goals we target are *fair* balancing and *cyclic* balancing. Fair balancing dictates that all bots should stand an equal chance of winning against all other bots. All bot win-rates should be 50%. Cyclic balancing dictates that some bots should stand a higher chance at winning against certain bots than against others. Torch bot should have a 70% win-rate against Nail bot, with the same applying to Nail bot against Saw bot and Saw bot against Torch bot. This is a relaxed form of Rock-Paper-Scissors, and will benefit a player able to guess which bot their opponent will choose, much as in deck selection in a deck-building game.

For these two experiments, the resulting game parameter vectors are shown in Table II. Given our own computational budget, we let the *fair* balancing and *cyclic* balancing experiments run for 260 iterations. We ran both experiments on consumer-end hardware, parallelizing at all times 6 different iterations or trials.

B. Computing an evaluation matrix

As with Rock Paper Scissors in Section IV, Workshop Warfare is a 2-player symmetric zero-sum game. This means that we can exploit the fact that the win-rate of two bots a and b is $w_{ab} = 1 - w_{ba}$. Let $\boldsymbol{\theta}$ denote an arbitrary parameter vector for Workshop Warfare. Let $w^{\boldsymbol{\theta}}_{ST}$ denote the win-rate of Saw bot vs Torch bot, Saw bot vs Nail bot $w^{\boldsymbol{\theta}}_{SN}$ and Torch bot vs Nail bot $w^{\boldsymbol{\theta}}_{TN}$. Our algorithm will attempt to find the right set of game parameters $\boldsymbol{\theta}$ that yields either a cyclic or fair balancing in terms of these win-rates.

To compute these win-rates, we simulate many head to head matches where each bot is controlled by an agent using MCTS to guide its actions. Each matchup's win-rates are computed from the result of 50 game simulations. A higher number of game simulations would result in a more accurate prediction of the true win-rate between two bots, at the cost of more computational time.

C. Monte Carlo Tree Search

A thorough description of MCTS is beyond the scope of this paper, see [15] for a comprehensive review. Here we use MCTS to create gameplaying agents[4] to auto-balance the game to meet our design goals.

MCTS relies on a forward-model of the game to run "internal" game simulations alongside the game being played. As such, it would not be possible to use MCTS on games for which a forward game model is not available, or for which the model is prohibitively slow. The MCTS agents we use could be replaced with any other method of creating gameplaying agents suitable for the game of concern, and are not an integral part of our method.

All MCTS agents use a computational budget of 625 iterations. A higher computational budget is directly related to a higher skill level [16]. Following this idea, our method could be used to balance a game at different levels of play by changing the computational budget.

We use a reward scheme that incentivizes bots to interact with one another by (1) giving negative score to actions that would increase distance between bots (2) giving positive / negative score to damaging the opponent / being damaged and (3) giving a score for winning the game. The magnitude of rewards (1), (2), and (3) varied between 0-10, 10-99, and 1000 respectively so as to represent a hierarchy of goals for the agent to follow.

[4]We have open-sourced our MCTS implementation: https://www.github.com/Danielhp95/Regym

VII. Results

Using Algorithm 1 defined in Section III, we found the following parameter vectors $\boldsymbol{\theta}_{fair}$ and $\boldsymbol{\theta}_{cyclic}$, corresponding to the meta-game balancing defined in Figure 6a and Figure 6b respectively. These parameter vectors are shown in Table II. We provide recordings of sample episodes for each balancing scenario[5]. As a game designer the most important question to ask is: *how do the different bots play?* We briefly describe the game parameterized under $\boldsymbol{\theta}_{fair}$ and $\boldsymbol{\theta}_{cyclic}$:

1) Fair balancing: Torch bot, with the most health (9), slowest movement (6) and lowest damage (3), plays like a tank[6]. Nail bot is a "glass cannon"; the fastest (2) and most damaging (7) character with the lowest health (4). Its cooldown of 1 tick allows it to quickly react to opponents close by, and to barrage other bots from a distance. Saw bot moves at a medium speed 4 and has to carefully approach opponents, but once it reaches them a victory is always guaranteed.

2) Cyclic balancing: Bot behaviours are similar to the previous case, with some differences. Saw bot is slower (5), often using the stand still action (S) to time movement to avoid damage. It exploits Torch bot's shorter range (3) and longer cooldown (5) to wait for an opening from a distance Nail bot, as fast as before but even more damaging (9) is able to position itself for a single nail that kills the slower Saw bot. Because Nail bot now has only 3 health, it dies to a single touch by Torch bot's flame, making it significantly weaker against it.

A. Discussion

Figure 7 shows how, as our algorithm iterated, both $\boldsymbol{\theta}_{fair}$ and $\boldsymbol{\theta}_{cyclic}$ generated game balacings which grew closer to the desired target balances. At the end of the 260 iterations, The balancing emerging from $\boldsymbol{\theta}_{cyclic}$ features an aggregated error of 9% win-rate over the target graph which we deem as acceptable. Unfortunately, the error associated with $\boldsymbol{\theta}_{fair}$ is large (16%) as the win-rate between Saw bot and Nail bot favoured Saw bot heavily, which we deem as unsatisfactory. However, given the downwards trend of Figure 7, we have reason to believe that better paramater vectors could be found, provided greater computational time.

For both experiments, each algorithmic iteration was completed, on average, every 20-25 minutes, and in total both experiments took approximately 96 hours each, where most of the computational time was spent by MCTS's internal simulations. This is evidence that our algorithm is computationally expensive. Although a linear speedup could be gained simply by increasing the number of CPUs, further improvements aimed reducing the computational load of the algorithm are needed to allow for the balancing of real-world games.

In Figure 7 between iterations 0 and 80 there are 8 iterations, more or less evenly spaced, which improve upon the best parameters found so far. Assuming each iteration takes 20 minutes, every 10 iterations, or roughly 3 hours and 20 minutes, our algorithm found game parameters that moved the

[5]Videos available at: https://github.com/Danielhp95/GGJ-2020-cool-game
[6]https://en.wikipedia.org/wiki/Tank_(video_games)

balancing closer to the designers' target balancing. This is a clear example of how our algorithm automates the balancing process. On the other hand, we also see in Figure 7 a gap between iterations 80 and 200 where our algorithm did not find a parameter vector which improved upon the current best solution. In wall-clock time, this gap took 40h. This is clearly an issue, specially for more computationally intensive games. From the user's perspective, our method does not return any information during those 40h because no new best parameter $\boldsymbol{\theta}$ was found. One is left to wonder if there are any metrics not directly relevant to the optimization process, which could be extracted from our algoritm's computation that may be of use to the game developers. This remains an open question.

Certain parameterizations might defy the original intent of the designer. In the field of AI, this is known as value misalignment. We name a few. In the fair balancing case, all bots can die from either 1 or 2 hits, which makes for short-lived matches. Furthermore, Saw bot's ability lasts for 6 ticks, whilst having a low cooldown of just 3 ticks. This makes its ability an almost permanent effect rather than a special action. As a parallel study, the playstyle displayed by MCTS tends to be very offensive at the beginning, and very defensive later on. More human-like methods for generating gameplaying agents would greatly benefit the result of our algorithm (line 5 of Algorithm 1), although we emphasize that this problem is orthogonal to our algorithm.

TABLE I: Optimized parameters of each bot type for fair and cyclic target graphs.

Bot Type	Parameter	Min	Max	Fair	Cyclic
Torch	Health	1	10	9	7
	Cooldown	1	6	3	5
	Damage	1	10	3	3
	Ticks between move	1	6	6	4
	Torch duration	1	6	3	2
	Torch range	1	4	4	3
Nail	Health	1	10	4	3
	Cooldown	1	6	1	1
	Damage	1	10	7	9
	Ticks between move	1	6	2	2
Saw	Health	1	10	6	4
	Cooldown	1	6	3	3
	Damage	1	10	2	6
	Damage change	1	10	7	6
	Ability duration	1	6	6	3
	Ticks between move	1	6	4	5

VIII. Related work

Quantitative methods for understanding games have been proposed in many forms. [17] presents a similar balancing overview to ours, introducing a generic iterative an automated balancing process. of sampling game parameters, using AI players (or real humans) and testing if a desirable balancing has been achieved. Our main differentiating contribution are balancing graphs as a designer friendly balancing description. Several strategies for the assessment of games without real player data are described in [18]. Our research most closely resembles the strategy defined as "Hypothetical player-testing", in which we are "trying to characterize only how a game operates with a particular player model" [18]. The forms that this type of hypothetical player-testing analysis can take are discussed in depth in [19], and specifically our work is concerned with the subcategory of quantitative analysis defined as Automated Analysis, helping designers evaluate (and often modify) their games without human playtests [19].

Machine Learning offers tools for automatic meta-game analysis. Harnessing existing supervised learning algorithms, [20] used random forests and different neural network architectures to assess meta-game balance by predicting the outcomes of individual matches using hand-crafted features that describe the strategies being used. The authors make an assessment of the overall balance of the meta-game by measuring "the prevalence of parallel strategies" [20], assuming balance to mean equal prevalence. While this is informative, it is predicated on a definition of balance that may not align with the goals of other game designers working on other projects, which may have definitions for balance that extend beyond prevalence. Additionally, such techniques are limited to assessing the current balance of a game context rather than providing a solution for balance issues that are discovered.

TABLE II: Win rates for Saw Vs. Torch (w_{ST}), Saw Vs. Nail (w_{SN}) and Torch Vs. Nail (w_{TN}) after balancing and their corresponding errors.

	Fair balance: $\boldsymbol{\theta}_{fair}$			**Cyclic balance: $\boldsymbol{\theta}_{cyclic}$**		
Match	Target	Found	Error	Target	Found	Error
w_{ST}	50%	50%	0%	70%	68%	2%
w_{SN}	50%	64%	14%	30%	36%	6%
w_{TN}	50%	52%	2%	70%	69%	1%

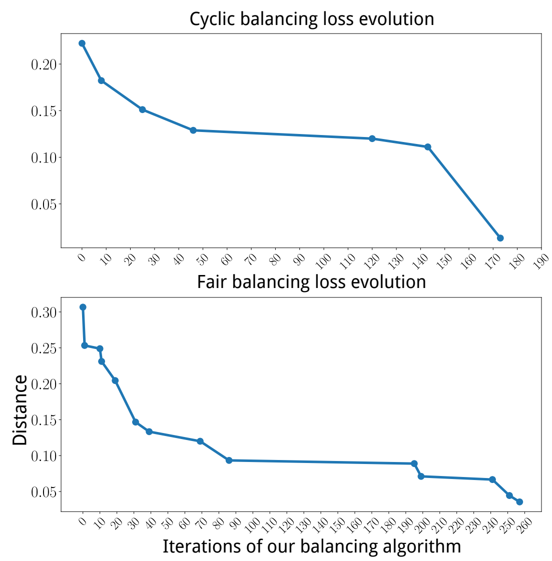


Fig. 7: Evolution of distance to target balance graph for Cyclic balancing (top) and fair balancing (bottom). Top graph stops at iteration 173, as it was the last iteration to improve upon the previous best parameter.

MCTS has been used for this type of simulation based analysis in the past, in [21] MCTS agents are used to model players at various skill levels in Scrabble and Cardonomicon to extract metrics that describe game balance. However, this type of analysis is concerned with the discovery of issues and takes no steps towards providing a solution to a balance problem once discovered.

The work by Liu and Marschner [22] uses the Sinkhorn-Knopp algorithm to balance a mathematical model, according to game theoretical constructs, representing a simplified version of the popular game Pokemon. In Pokemon, each pokemon type[7] has advantages and disadvantages against various other types. The authors tune these type features to make them all equally viable pokemon types. This is akin to our fair balancing experiment in Section VI. This approach concerns itself with mathematical comparisons between strategies based on an existing table of matchup statistics, which may not exist for most games, especially those still in development.

Leigh et al. [23] used co-evolution to evolve optimal strategies for CaST, a capture the flag game. Populations of agents were evolved in an environment with a set of game parameters. The distribution of the resulting agents across simplex heat maps of different strategies was used to assess whether or not the game was balanced with those game parameters by considering balance to be a situation where any core strategy should beat one of the other core strategies and lose against another, similar to our definition of *cyclic* balancing. They manually modified play parameters and iterated to find a configuration with a desirable heatmap. Our approach builds upon this work by automating the manual parameter adjustment stage, it also broadens the definition of balance by allowing the designer to specify exactly what meta-game state they consider balanced.

IX. CONCLUSION

In this paper we present an algorithm to autobalance a game as requested by a designer. We do this by combining concepts from AI for gameplaying, optimization, game and graph theory. We also develop the mathematical foundation for this tool, demonstrating its empirical convergence in a simple toy domain and showcasing its potential in a richer game environment. To our knowledge, our work is one of the first steps in the field of game balancing towards robust tools for automated balancing in multiagent games. The issues of computational time, non human-like AI behaviour and the complexity of generating gameplaying agents remain as obstacles in the path towards accessible adoption of our algorithm by designers.

Our contributions could be transformed into the "backend" of an actual tool. To make it amenable to be used by non-technical individuals, a user-friendly "frontend" should be developed, exposing an interface to (1) parameterize a game and (2) make it easy to specify a level of abstraction and its corresponding balance graph.

[7] An overview of Pokemon types: https://bulbapedia.bulbagarden.net/wiki/Type

REFERENCES

[1] N. Shaker, J. Togelius, and M. J. Nelson, *Procedural content generation in games*. Springer, 2016.

[2] C.-S. Lee and I. Ramler, "Investigating the impact of game features on champion usage in league of legends." in *FDG*, 2015.

[3] M. Carter, M. Gibbs, and M. Harrop, "Metagames, paragames and orthogames: A new vocabulary," in *Proceedings of the international conference on the foundations of digital games*, 2012, pp. 11–17.

[4] S. Omidshafiei, C. Papadimitriou, G. Piliouras, K. Tuyls, M. Rowland, J.-B. Lespiau, W. M. Czarnecki, M. Lanctot, J. Perolat, and R. Munos, "{\alpha}-rank: Multi-agent evaluation by evolution," *arXiv preprint arXiv:1903.01373*, 2019.

[5] D. Balduzzi, M. Garnelo, Y. Bachrach, W. Czarnecki, J. Perolat, M. Jaderberg, and T. Graepel, "Open-ended Learning in Symmetric Zero-sum Games," 2019.

[6] M. P. Wellman, "Methods for empirical game-theoretic analysis," in *AAAI*, 2006, pp. 1552–1556.

[7] X. Gao, B. Xiao, D. Tao, and X. Li, "A survey of graph edit distance," *Pattern Analysis and applications*, vol. 13, no. 1, pp. 113–129, 2010.

[8] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," in *Advances in Neural Information Processing Systems 24*, J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2011, pp. 2546–2554. [Online]. Available: http://papers.nips.cc/paper/4443-algorithms-for-hyper-parameter-optimization.pdf

[9] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," *CoRR*, vol. abs/1907.10902, 2019. [Online]. Available: http://arxiv.org/abs/1907.10902

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

[11] R. A. Bartle, "Player types," 2008.

[12] D. Hernandez, K. Denamganaï, Y. Gao, P. York, S. Devlin, S. Samothrakis, and J. A. Walker, "A generalized framework for self-play training," in *2019 IEEE Conference on Games (CoG)*. IEEE, 2019, pp. 1–8.

[13] G. N. Yannakakis and J. Togelius, *Artificial intelligence and games*. Springer, 2018, vol. 2.

[14] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[15] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of {Monte Carlo Tree Search} methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012.

[16] P. L. Lanzi, "Evaluating the complexity of players' strategies using mcts iterations," in *2019 IEEE Conference on Games (CoG)*. IEEE, 2019, p. 1–8.

[17] M. Beyer, A. Agureikin, A. Anokhin, C. Laenger, F. Nolte, J. Winterberg, M. Renka, M. Rieger, N. Pflanzl, M. Preuss *et al.*, "An integrated process for game balancing," in *2016 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 2016, pp. 1–8.

[18] M. J. Nelson, "Game metrics without players: Strategies for understanding game artifacts," in *Artificial Intelligence in the Game Design Process*, 2011.

[19] A. B. Jaffe, "Understanding game balance with quantitative methods," Ph.D. dissertation, 2013.

[20] R. Argue, "Supervised learning as a tool for metagame analysis," 2014.

[21] A. Zook, B. Harrison, and M. O. Riedl, "Monte-carlo tree search for simulation-based play strategy analysis," in *FDG*, 2015.

[22] A. J. Liu and S. Marschner, "Balancing zero-sum games with one variable per strategy," in *AIIDE*, 2017.

[23] R. Leigh, J. Schonfeld, and S. J. Louis, "Using coevolution to understand and validate game balance in continuous games," in *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, 2008, pp. 1563–1570.