Efficient Evolutionary Methods for Game Agent Optimisation: Model-Based is Best

Simon M Lucas[1], Jialin Liu[2], Ivan Bravi[1], Raluca D. Gaina[1], John Woodward[1], Vanessa Volz[1], Diego Perez-Liebana[1]

[1]Queen Mary University of London, London, UK

[2]Southern University of Science and Technology, Shenzhen, China

simon.lucas@qmul.ac.uk, liujl@sustc.edu.cn, i.bravi@qmul.ac.uk, r.d.gaina@qmul.ac.uk, j.woodward@qmul.ac.uk, v.volz@qmul.ac.uk and diego.perez@qmul.ac.uk

Abstract

This paper introduces a simple and fast variant of Planet Wars as a test-bed for statistical planning based Game AI agents, and for noisy hyper-parameter optimisation. Planet Wars is a real-time strategy game with simple rules but complex gameplay. The variant introduced in this paper is designed for speed to enable efficient experimentation, and also for a fixed action space to enable practical inter-operability with General Video Game AI agents. If we treat the game as a win-loss game (which is standard), then this leads to challenging noisy optimisation problems both in tuning agents to play the game, and in tuning game parameters.

Here we focus on the problem of tuning an agent, and report results using the recently developed N-Tuple Bandit Evolutionary Algorithm and a number of other optimisers, including Sequential Model-based Algorithm Configuration (SMAC). Results indicate that the N-Tuple Bandit Evolutionary Algorithm offers competitive performance as well as insight into the effects of combinations of parameter choices.

1 Introduction

Games provide a rich and natural source of noisy optimisation problems. Uncertainty arises in many ways in the design, testing and application of game-playing agents, and in the design optimisation of games to meet desired criteria. Some games are inherently stochastic due the roll of dice, shuffle of cards or random number generation, but other sources of uncertainty include the unpredictable actions of an opponent, the use of inherently stochastic agents, and the results of human play-testing of games.

Regardless of the purpose of the optimisation, there is a natural desire to make the optimisation as efficient as possible. This efficiency enables agents and game parameters to be optimised rapidly, allowing the use of such optimisers to be more widespread. Extremely rapid optimisation could even spur the development of new game design tools, where a designer could be given immediate advice on the likely effects of changes to the game design, as the optimiser explores the most interesting choice of game parameters.

It is widely known in machine learning and algorithm optimisation that using a good choice of parameters can improve performance by an order of magnitude or more, compared to a poor choice. On the game design side, well chosen parameters can be the difference between an excellent game and an unplayable one. Within the game AI community, evolutionary algorithms are a very common choice of optimisation method (Yannakakis and Togelius 2018). However, there are often significant advantages in using a more sophisticated model-based evolutionary approach, and we provide an example of this below.

We designed and implemented a simplified version of Planet Wars, which is itself a relatively simple (relative to a game like StarCraft for example) but interesting real-time strategy game. Although Planet Wars has been used as the basis for a very successful Google AI challenge, the software behind that is complex and not fast enough for the practical application of statistical forward planning algorithms such as Monte Carlo Tree Search and Rolling Horizon Evolution. To enable the use of these algorithms and also allow rapid experimentation we designed and implemented a simplified version of the game. This cut-down version provides a good benchmark for game AI agents in that it has sufficient skill-depth to expose a wide span of player abilities, but runs very quickly, at more than 10 million game ticks per second. Evidence of skill-depth will be presented in Section 5.

The version described in this paper is as simple as we could make it while retaining some of the essence of the game. A fast and full-featured re-implementation of Planet Wars that adds gravity and a variety of actuators is described by Lucas (Lucas 2018), though that version is an order of magnitude slower than the version described here. In this paper we focus on optimising game-playing agents, and leave optimising the game parameters to meet specified design criteria for future work.

Another motivation behind this work is to provide an additional set of games for the General Video Game AI (GVGAI) competition framework. While most GVGAI games to date have been implemented in the Video Game Description Language (VGDL), there are advantages to writing games in a language such as Java, as done for this paper, as noted in (Perez-Liebana et al. 2018). Java provides more flexibility in designing game rules that would be difficult to express in VGDL, and also enables much faster execution of the game. In addition to the implementation of the game, we also provide a wrapper class so that GVGAI agents see the standard GVGAI interface, and can play the game without

AAAI-2019 Workshop on Games and Simulations for Artificial Intelligence.

any modification. The Planet Wars variant developed here is suitable for either the single player or two player planning tracks, and has already been tested with the GVGAI framework. With some interface wrapping it could also be used for the GVGAI learning tracks. The development of games with strategic depth for GVGAI and for Atari Learning Environment (ALE) agents will provide a fresh range of challenges for general video game playing.

Having set up the game with an agent configuration problem as a noisy optimisation problem, the main contributions of the paper are to show the effectiveness of model-based optimisation approaches compared to non model-based evolution, and also to provide insight in to how the N-Tuple Bandit Evolutionary Algorithm (NTBEA) operates.

The rest of the paper is structured as follows. Section 2 presents the algorithms considered in this work for optimizing the hyper-parameters. Section 3 describes variations of the Planet Wars game. Section 4 describes the test problem. Section 5 presents the experimental results and discusses the results. Section 6 concludes.

2 Hyper-Parameter Optimisation

In this section we describe the main approaches to handling hyper-parameter optimisation, specifically for noisy applications.

2.1 Main Approaches

The performance of an algorithm may be highly dependent on its parameter settings, which is why hyper-parameter optimisation, i.e. finding optimal parameter configurations for optimisation algorithms, is a popular topic of research. In hyper-parameter optimisation, especially in context of real-world problems with already expensive fitness functions, the evaluation of a single parameter configuration is usually computationally expensive. For this reason, hyper-parameter optimisation algorithms often use model-based approaches (Hutter, Hoos, and Leyton-Brown 2011; Bartz-Beielstein 2006). These algorithms train surrogate models on evaluated solutions and use the obtained information to reduce the number of evaluations required in order to avoid prohibitively long algorithm runtimes. Several methods to integrate the information have been proposed (Jin 2011).

The existence of noise adds another dimension to hyper-parameter optimisation, as it significantly affects the performance of an optimisation algorithm and may require special coping measures to be taken, such as selective resampling as done by the N-Tuple Bandit Evolutionary Algorithm.

2.2 N-Tuple Bandit Evolutionary Algorithm

The N-Tuple Bandit Evolutionary Algorithm (NTBEA) was formalised by Lucas *et al.* (Lucas, Liu, and Perez-Liebana 2018) in 2018, though an earlier version has been used for evolving game parameter settings (Kunanusont et al. 2017). The NTBEA combines evolution with bandit-based sampling and follows on from the bandit-based Random Mutation Hill Climber proposed previously (Liu, Perez-Liebana, and Lucas 2017). The NTBEA was designed for hyper-parameter optimization, and has been tested on evolving new game instances by optimizing game parameter settings (Kunanusont et al. 2017), on off-line optimization of parameter settings for game-playing AI agents on different video games (Lucas, Liu, and Perez-Liebana 2018) and on on-line learning of parameter settings for an MCTS agent on games in the General Video Game AI (GVGAI) framework (Sironi et al. 2018).

For noisy game optimisation problems the NTBEA has a number of attractive features:

- Rapid convergence to good solutions in cases of high noise and small evaluation budgets
- Informative and intuitive model provides statistics to explain parameter choices
- Low computation overhead compared to Gaussian Processes models, for example
- Computation overhead scales well for large search spaces and large evaluation budgets due to efficient constant-time data structures, namely arrays or hash-maps
- The algorithm is relatively simple, making it easy to port to and embed in new platforms

Currently there exist open source implementations of the NTBEA in Java[1] and in Python[2].

2.3 Sequential Model-based Algorithm Configuration

For comparison, we consider one of the most popular hyper-parameter optimization algorithms, Sequential Model-based Algorithm Configuration (SMAC), proposed by Hutter *et al.* (Hutter, Hoos, and Leyton-Brown 2011).

SMAC is a model-based algorithm configuration tool and is an extension of Sequential Model-Based Optimization (SMBO). The algorithm alternates between constructing predictive models and utilising them to determine which parameter configurations to choose next. This approach is described by the authors as a balance between intensification and diversification (similar to exploitation and exploration). We use the SMAC version 3 developed by the ML4AAD Group of the University of Freiburg[3] using Python and C++.

3 Planet Wars

3.1 Google AI Challenge and Dagstuhl AI Hackathon

Planet Wars was the subject of the Google AI challenge in 2010 by the University of Waterloo in Canada (Fernández-Ares et al. 2011). The game is a simple real-time strategy game that is fun for humans to play and provides an interesting challenge for AI.

According to Buro et al. (section 4.1 in (Lucas et al. 2015)), who used Planet Wars in a hackathon, the game benefits from a simple rule set and a real-time decision complexity. They also observed that the particular implementation

[1] https://github.com/SimonLucas/ntbea

[2] https://github.com/Bam4d/NTBEA

[3] SMAC on GitHub: https://github.com/automl/SMAC3.

was rather slow and only able to play of the order of one game per second, which made optimising the AI players a time-consuming process.

3.2 Fast Planet Wars Variants

The approach we took to developing variants of Planet Wars was to strip it back to the bare minimum of essential features which would still allow the game to be recognisable as a planet invasion game. We began with the following aims:

- The game should be fast, allowing rapid copying of the game state, and even more rapid advancing of the game state given a set of selected actions (i.e. the nextState function). We aim to run the game at more than one-million ticks per second on a typical modern laptop. This makes it suitable for testing statistical forward planning algorithms.
- Make the game easily scalable to a large number of planets, without changing the structure of the agents or making the game too slow. Ideally the computational cost of the next-state function should be independent of the number of planets, or at worst be linear in the number of planets.
- Ensure the game has significant skill-depth (this can be estimated by the span of Elo ratings (or more simply, league table scores), between strong and weak agents).
- Make the code simple and easily extensible to encourage others to copy and adapt.
- Retain the simultaneous real-time aspect of the game.
- Make the game playable with a small fixed-size action space, suitable for General Video Game AI agents, or Atari Learning Environment (ALE) agents.

A screen shot of our simplified *Planet Wars*Game is shown in Figure 1, and was first described by Lucas *et al.* (Lucas, Liu, and Perez-Liebana 2018). The game well exceeds our target speed and runs at more than 10 million game ticks per second.

We first reiterate the main features of the game from (Lucas, Liu, and Perez-Liebana 2018) before describing some aspects in more detail:

- There are no neutral planets: the ships on each planet are either owned by player 1 or player 2.
- At each game tick, a player moves by shifting ships to or from a player's ship buffer, or by moving its current *planet of focus*.
- When a player transfers ships it is always between its buffer and the current planet of focus.
- At each game tick the score for each player is the sum of all the ships on each planet it owns, plus the ships stored in its buffer. We have two versions of the game: the easiest for the planning agents, and the one used for most of the experiments in this paper, also adds in the ships in a player's buffer to its score, a more deceptive version of the game does not include this.

Our intuition is that two of the elements missing from the cut-down version will most likely significantly reduce the skill-depth: these are the time-delay between the ships leaving their origin and arriving at their destination, and the neutral planets. Neutral planets pose an interesting dilemma for a player: invading them wastes a player's ships, unlike invading the opponents planets where the ships lost directly deplete the opponent's total. But invading neutral planets is also necessary in order to grow a player's ship producing capacity. Therefore a common ploy is to wait for the opponent to invade a neutral planet, then issue an immediate strike on it. Hence, it will be interesting future work to compare the skill-depth of the version described in this paper with the version described in (Lucas 2018).

3.3 Game Play

At each game step a player executes one of 5 actions. Do nothing, move planet of focus (clockwise or anti-clockwise), or attempt to move planets from buffer to planet of focus, or vice versa.

To make the agent evaluation process as efficient as possible the game used in all our experiments was played for a fixed budget of 200 game ticks, as most games were already decided by this point. The objective of each player is to have the higher score when the game ends, which is calculated as the sum of all the ships on each of their planets plus the ships stored in their buffer.

In addition to the agent interface, the game is also playable via the arrow-keys on a keyboard. The lead author has played many games against the best evolved agents in this study using the best configurations specified below, and is rarely able to beat them when playing at one second per move. However, the circular layout of the planets is confusing for left/right key control (just as playing Atari's Tempest using a keyboard is confusing).

4 Optimising Rolling Horizon Evolutionary Planning Agents

We compared the NTBEA and SMAC (presented in Section 2), as well as several non-model based evolutionary algorithms, on optimizing hyper-parameters of a game playing agent on the Fast Planet Wars game.

4.1 Agents

In this work, we optimise the parameters of a rolling horizon $(1 + 1)$-Evolutionary Algorithm, denoted as RHEA in the rest of the paper. In the RHEA agent, the individual is an action sequence of a fixed horizon. The algorithm is initialised by creating a population of random actions sequences or rollouts by sampling uniform randomly from the available set of actions. The choice of evolutionary algorithm optimiser is one of the parameters of the RHEA agent. For all these experiments we used a random mutation hill climber, also known as a $(1 + 1)$ EA. In this case the algorithm is initialised with a single random rollout.

The key parameters to be optimised for this agent and their legal values are summarised in Table 1. The notations are detailed as follows:

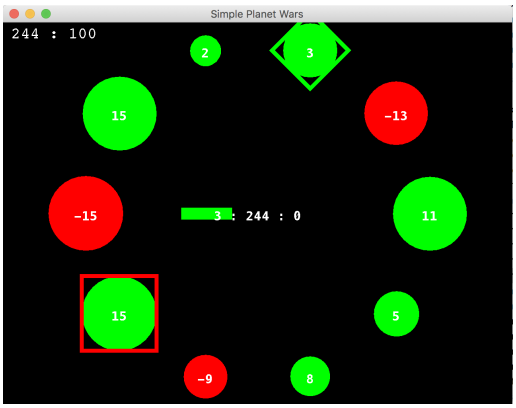

Figure 1: A simplified version of planet wars made for speed. This is a competitive two player game played by the green player versus the red player. In this version there are no neutral planets, the initial state randomized regarding planet ownership and size. All transfers between planets go via a player's respective buffer in the middle of the area, and are transferred between the buffer and the planet of focus, identified by the green diamond and the red square. The aim for each player is to acquire the most ships within a specified time limit or to win the game by occupying all the planets.

Table 1: Search space of the parameter settings.

Variable	Type	Legal values
$nbMutatedPoints$	Integer	$0.0, 1.0, 2.0, 3.0$
$flipAtLeastOneBit$	boolean	$false, true$
$useShiftBuffer$	boolean	$false, true$
$nbResamples$	Integer	$1, 2, 3$
$sequenceLength$	Integer	$5, 10, 15, 20, 25, 30$

- $nbMutatedPoints$ defines the mutation probability, how likely a mutation occurs at every dimension, by dividing it by the number of dimensions (in this case the number of dimensions is the same as the sequence length)
- $flipAtLeastOneBit$ indicates if at least one mutation should occur at each time;
- $useShiftBuffer$ enabled or not: if disabled, at any time step $t+1$, the initial individual is reset to random, otherwise, the individual at time step t shifts its action sequence forward and fills the last position by a random action;
- $nbResamples$ defines how many time the individual (i.e. the action sequence) is re-evaluated as the game is stochastic;
- $sequenceLength$ defines the planning horizon (i.e. the length of each action sequence / rollout).

As *Planet Wars* is a two-player game without win or loss, only the game scores for both players are reported at the end of a game. We define the following evaluation function for the agent, assuming optimizing for player 1:

$$f(player_1, player_2) = \begin{cases} 1, & \text{if } score_1 > score_2 \\ -1, & \text{otherwise} \end{cases} \quad (1)$$

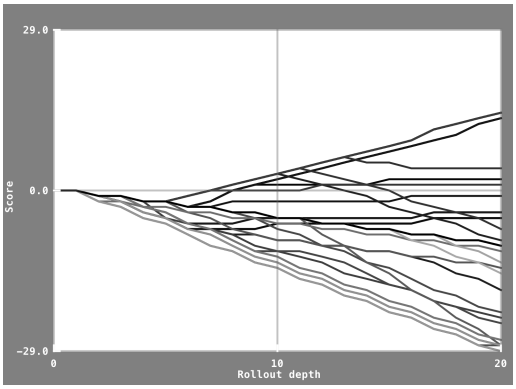

Figure 2: The graph provides insight into a rolling horizon evolutionary algorithm playing Planet Wars. The RHEA agent has a rollout length (horizon) of 20, and this variant of the game includes the ships on their buffer in each agent's score.

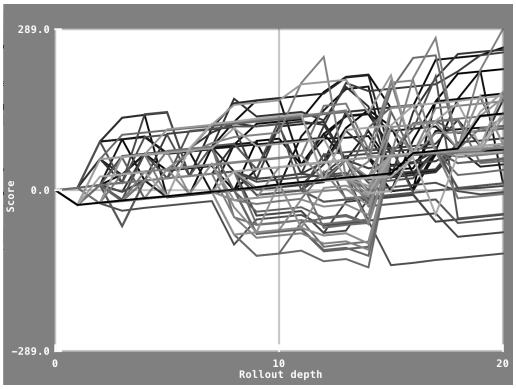

Figure 3: This graph is similar to the one in figure 2 except that the game score does not include the ships in each buffer, meaning that agents need to see beyond the apparent loss in score that moving ships to their buffer would entail. This causes short-term agents to perform very badly compared to the previous version of the game.

where $score_1$ and $score_2$ denotes the final scores obtained by $player_1$ and $player_2$. The RHEA agent aims at maximising the game score from the perspective of its player. Since the player with the largest number of ships wins, this is a sensible objective to optimise.

To give an idea of what each agent sees during its rollouts, we plot the score difference from the current game state to 20 game ticks ahead for a typical game-state. Figure 2 shows the results of the rollouts when we include the ships in the buffers in each player's score. To give an idea of how simple changes to the game can affect an agent's view of the world, Figure 3 shows how the score typically varies when the buffers are not included in the score. Note that this simple change makes the game noisier and more deceptive, and harder for the short-term agent to play. Figure 4 shows the game score traces when playing these two agents against each other for these two game variants. Not including the

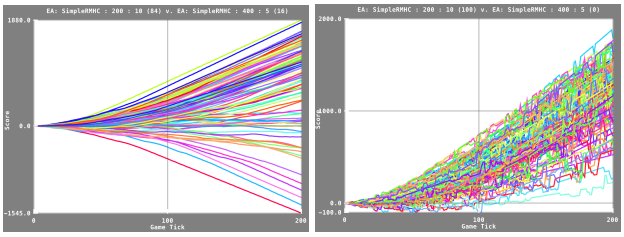

Figure 4: The figure shows a medium-term planning agent (200 rollouts of length 10) playing 100 games against a short term agent (400 rollouts of length 5). The number of rollouts per game state is adjusted to keep the tick-budget per action constant at $2,000$. Each line shows how the score developed over each game tick. Left: Buffers are included in the score, Right: Buffers not included. With the buffer, we observe the short-term agent being convincingly beaten, but still winning 16 games. Without the buffer, the short-term agent lost all 100 games. This illustrates how small changes to the game can have significant effects on the agents that play it.

buffered ships in the score has a significant effect both on the game outcomes (the short-term agent now loses all 100 games) and the smoothness of the score trajectories.

5 Results and discussion

In this section we describe the experimental settings followed by a more detailed analysis of the optimisation problem. The comparison with other evolutionary algorithms is presented in Section 5.4.

5.1 Experimental settings

The game is played by a RHEA with settings selected by an optimizer versus a RHEA agent with manual tuned parameter setting $(1, true, true, 1, 5)$ without knowledge about the fitness landscape denoted as A_{fixed}. At every game tick, each of the agents has $2,000$ forward model calls as its simulation budget. The fitness value reported for a solution is $f(A_{tuned}, A_{fixed})$ with f defined in (1).

The opponent model used by each agent was fixed for all the hyper-parameter optimisation experiments reported below. In order to identify the baseline performance, we used an agent that does not act at all (i.e. returns a *do nothing* action each time) both for the agent being tuned, and for its fixed opponent. For the analysis visualised in Figure 4 we used a random agent instead.

We used the default settings for each of the optimisation algorithms tested. In addition, we experimented with the exploration settings (parameters k and ϵ) in NTBEA in order to gain a better understanding of the algorithm. The results are reported in Section 5.4.

5.2 Fitness Distribution

Figure 5 illustrates the performance of the agent using the 288 possible parameter settings described in Table 1, sorted by the average fitness value over 100 trials (i.e. the RHEA agent with each set of parameters player 100 games against a fixed opponent to evaluate its fitness). The blue shadow shows one standard error either side of the mean. Due to the inherent noise in the evaluation process, the parameters that appear as optimal could vary a bit from run to run, but there are very clear differences between different sets of players, with the best parameter settings performing much better than the worst ones. We consider the fact that different agents play the game with a range of clearly separable abilities as evidence of skill-depth.

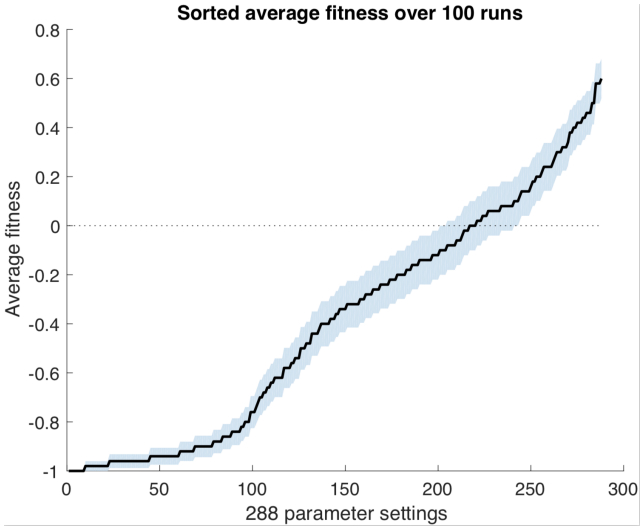

Figure 5: Sorted average fitness obtained by the agent using the 288 legal parameter settings. Each of the parameter settings is tested 100 times. Higher score fitness refers to being stronger than the opponent. The dashed horizontal line refers to when the parameterised player has identical performance to the fixed opponent.

The best parameter setting, according to the 100 tests made of every point in the search space is $(3, true, true, 1, 15)$ with average fitness 0.6. As shown in Figure 5, there is a clear difference in performance between many of the different parameter settings, though also some which are not clearly separated from each other.

5.3 Observing the N-Tuple Statistics

Figure 6 shows how fitness varies across a number of 2-tuple parameter choices, again when averaged over all 288 points in the search space, each evaluated 100 times. Clearly there are dependencies between the parameter combinations. Note how most pairwise combinations have an average value of below zero (averaged over all points sampled). This is due to the fact that most parameter settings (more than 200 of the 288 possible as shown in Figure 5) score below zero.

To understand the estimation of parameters by NTBEA optimizer, we plot in Figure 7 the average fitness for different sequence lengths evaluated by the NTBEA in the worst optimisation trial and a randomly selected successful trail (optimum found) using a low budget (288 game evaluations) and a higher budget (2880 game evaluations). As reference, the average fitness values for different sequence length over

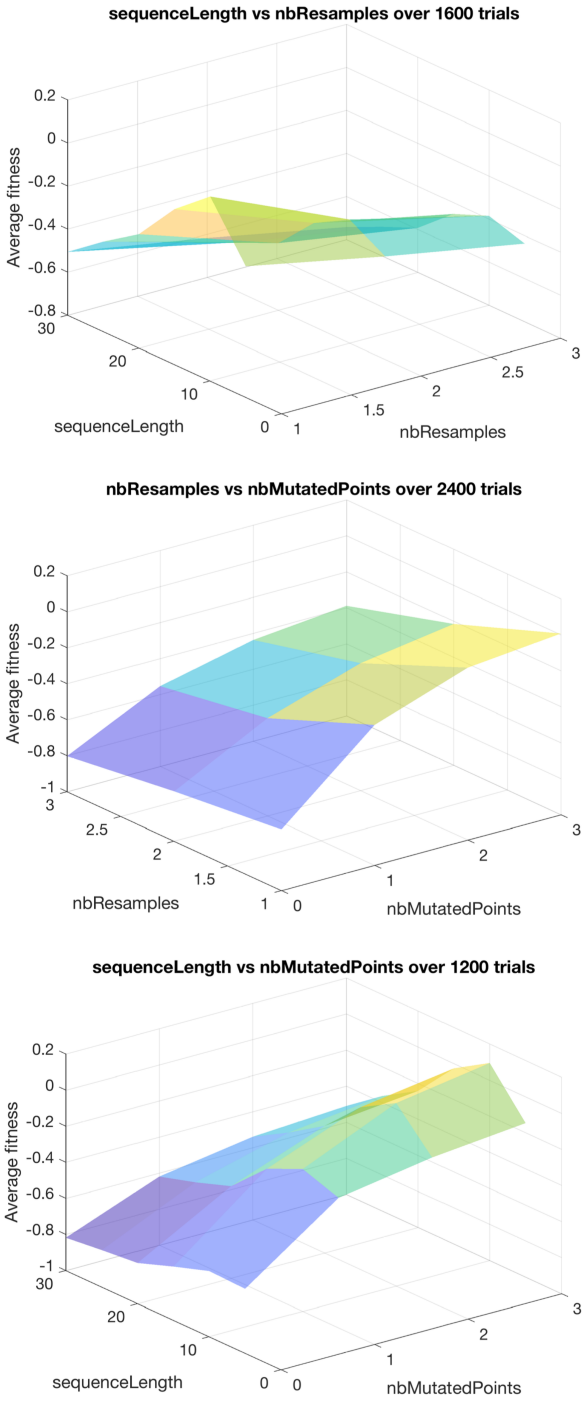

Figure 6: The impact of number of resamplings, mutation rate and the planning horizon (sequence length). z-axis shows the average fitness when fixing values for two of the parameters, as shown in x-axis and y-axis.

100 repetitions of games of the agent using all the 288 possible parameter settings are illustrated using the black solid curve.

Interestingly, the graph shows big differences in how fitness varies with respect to sequence length depending on how the space has been sampled, with some runs even incorrectly indicating that a length of 5 is best. The best solutions involve a sequence length of either 10 or 15, and this usually emerges from the statistics given enough samples. Note though that the large sample budget runs with the high peaks differ greatly from the overall statistics (black line).

The individual runs show how the NTBEA samples fitter regions of the space over the course of a run, which is especially clear in the large positive spikes for the 2880 budget.

With sufficient budget the NTBEA is able to provide a better estimation of the performance achieved by different parameter values, and note how tight the error bars have become for the best sequence lengths of 10 or 15 for the large 2880 budget runs, compared the error bars for worse performing sequence lengths on those runs.

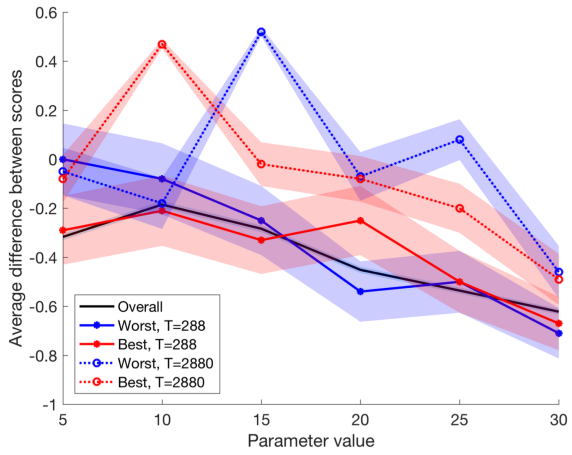

Figure 7: Average fitness for different sequence length evaluated by the NTBEA optimizer. The black solid curve shows the average fitness over 100 repetitions of games of the agent using all the 288 possible parameter settings, and is used as a reference. The solid and dashed red curves show the results of one of the successful trials (optimum found) using $T = 288$ and $T = 2880$ as budget. The solid and dashed blue curves show the results of the worst trial using $T = 288$ and $T = 2880$ as budget.

5.4 Performance Comparison

The performance results of several evolutionary algorithms are shown in Table 2, with brief notes below.

- The Random Mutation Hill Climber (RMHC) is the simplest evolutionary algorithm, also called a (1+1) EA. Despite its simplicity it often performs well. In this case though it performs poorly due to the high levels of noise. RMHC(1) uses no resampling, while RMHC(5) resamples each candidate 5 times to alleviate the noise, but at

Table 2: Comparison of different optimisation algorithms given 288 game evaluations as budget, except where stated (5x and 10x indicate 5 times and 10 times the standard budget respectively; this was only used for CMA-ES.)

Algorithm	Avg. ± StdErr
RMHC(1)	-0.29 ± 0.01
RMHC(5)	0.01 ± 0.01
CMA-ES(1x)	-0.12 ± 0.04
UH-CMA-ES(1x)	-0.09 ± 0.04
SGA	0.03 ± 0.01
CMA-ES(5x)	0.29 ± 0.02
UH-CMA-ES(5x)	0.32 ± 0.03
SWcGA	0.36 ± 0.01
SMEDA	0.38 ± 0.01
NTBEA(1,2,5)-	0.41 ± 0.01
UH-CMA-ES(10x)	0.44 ± 0.02
CMA-ES(10x)	**0.48 ± 0.02**
SMAC	**0.49 ± 0.01**
NTBEA(1)	**0.50 ± 0.01**
NTBEA(1,2)	**0.51 ± 0.01**
NTBEA(1,2,5)+	**0.51 ± 0.01**

the cost of wasted evaluations (the total sample budget was limited to 288).

- The Simple Genetic Algorithm (SGA) also performs poorly, in this case due to its inefficient use of the small sample budget.
- We included CMA-ES due to its high performance across a wide range of problems. CMA-ES operates in continuous search spaces, so to apply it to these discrete problems we used a box constraint in the range 0 to 1 in each dimension, and discretized the continuous value in each dimension by dividing into equal size ranges within the unit interval. We also tried the CMA-ES with uncertainty handling (Hansen et al. 2009) which is meant to be better suited to noisy problems, but did not improve on standard CMA-ES, perhaps due to the low sampling budget. CMA-ES(5x) and CMA-ES(10x) indicate that we modified the fitness function to use 5 times or 10 times resampling: these used up to 10 times larger sampling budget (10x indicates that 2,880 games were played during the optimisation).
- Sliding Window Compact GA (SWcGA) and SMEDA (Sliding Mean EDA) are new versions of the Compact GA (cGA) designed to be more sample efficient by incorporating a sliding window of candidate solutions. The SWcGA is described in (Lucas, Liu, and Pérez-Liébana 2017). They both use models to improve sample efficiency, though the models only estimate which parameter values are most likely to be optimal. In contrast, NTBEA and SMAC estimate the expected fitness of each parameter setting, and NTBEA goes one step further to also estimate the variances of each parameter setting.

All model based approaches greatly outperformed the non-model based alternatives in this study. All the approaches with results on boldface are better than the non-bold ones with statistical significance, but are not significantly different to each other. We ran several version of the NTBEA with different tuning. The numbers in parentheses show the n-tuples which were used for each configuration of the NTBEA. The poor performance of the NTBEA(1,2,5)− is due to a particular problem we observed. Depending on the particular combination of k and ϵ it is possible for the NTBEA to attempt to sample every point in the search space at least once. This is not a problem for the smaller n-tuples, and for NTBEA(1,2,5)+ we fixed it by setting ϵ to 0.5 and k to 1.0. See (Lucas, Liu, and Perez-Liebana 2018) for an explanation of these parameters.

6 Conclusion and further work

In this paper we explored noisy optimisation in the context of a simple and fast version of Planet Wars which offers a good degree of skill-depth and is compatible with GVGAI agents. For future work it would be interesting to compare the skill-depth of this game with the existing GVGAI games, both for the single and two-player versions. For the optimisation results in this paper we used a fixed opponent which means the game can then be treated as a single-player game, the difficulty of which depends on the strength of the fixed opponent. Tuning an agent to win a single-player game can be treated as a standard optimisation problem, though in this case an exceptionally noisy one due to the random initial game states and the stochastic nature of the agents.

The results clearly demonstrate how the use of model-based algorithms (both NTBEA and SMAC) were able to outperform non model-based alternatives, such as a random mutation hill climber and a simple genetic algorithm. One of the main messages of this paper is that games are a natural source of noisy optimisation problems, and that better performance is often obtained by using model-based approaches to deal with the noise.

NTBEA and SMAC offered similar performance, but NTBEA has the advantage of providing more informative output, with detailed statistics for each parameter choice in each dimension and in each combination of dimensions modelled by the n-tuples (see Figure 7). NTBEA also offers explicit control over how exploitative versus how explorative the algorithm should be.

It would be interesting to compare NTBEA to SMAC on optimizing other algorithms or other games, such as tuning an AI agent for General Video Game Playing. Though NTBEA has been applied to tune an MCTS agent for General Video Game AI (Sironi et al. 2018) and General Game Playing (Sironi and Winands 2017), it was not compared to SMAC or CMA-ES. The recently released NeverGrad toolbox also provides a set of optimisers for further comparisons (Rapin and Teytaud 2018).

The choice of tuples used in NTBEA is naively selected here, and also little effort was made to tune the other parameters of the NTBEA, which are the exploration constant k and the progressive widening parameter epsilon. The selection of tuples with a-priori knowledge of the agent to be tuned could further increase the efficiency of the optimisation. Naturally a high-level NTBEA could be used to tune an NTBEA running for a specific problem.

The paper provides further evidence of the importance of parameter tuning. The win-rates of the agents evaluated in Figure 4, show how a tuned agent beat an untuned agent 84 games to 16 in one case and 100 games to zero in the other case. Parameter tuning is important, and model-based methods often provide the most efficient approach, especially when the objective function is noisy.

Finally, it is worth emphasizing that the NTBEA does not just provide useful statistics on the best combinations of parameter, but actively uses those statistics to inform every decision about which point in the search space to sample next during a run.

References

Bartz-Beielstein, T. 2006. *Experimental Research in Evolutionary Computation – The New Experimentalism*. Springer.

Fernández-Ares, A.; Mora, A. M.; Merelo, J. J.; García-Sánchez, P.; and Fernandes, C. 2011. Optimizing player behavior in a real-time strategy game using evolutionary algorithms. In *Evolutionary Computation (CEC), 2011 IEEE Congress on*, 2017–2024. IEEE.

Hansen, N.; Niederberger, A. S. P.; Guzzella, L.; and Koumoutsakos, P. 2009. A method for handling uncertainty in evolutionary optimization with an application to feedback control of combustion. *IEEE Transactions on Evolutionary Computation* 13(1):180–197.

Hutter, F.; Hoos, H. H.; and Leyton-Brown, K. 2011. Sequential model-based optimization for general algorithm configuration. In *International Conference on Learning and Intelligent Optimization*, 507–523. Springer.

Jin, Y. 2011. Surrogate-assisted evolutionary computation: Recent advances and future challenges. *Swarm and Evolutionary Computation* 1(2):61–70.

Kunanusont, K.; Gaina, R. D.; Liu, J.; Perez-Liebana, D.; and Lucas, S. M. 2017. The n-tuple bandit evolutionary algorithm for automatic game improvement. In *2017 IEEE Congress on Evolutionary Computation (CEC)*.

Liu, J.; Perez-Liebana, D.; and Lucas, S. M. 2017. Bandit-based random mutation hill-climbing. In *2017 IEEE Congress on Evolutionary Computation (CEC)*.

Lucas, S. M.; Mateas, M.; Preuss, M.; Spronck, P.; and Togelius, J. 2015. Artificial and computational intelligence in games: Integration (dagstuhl seminar 15051). In *Dagstuhl Reports*, volume 5. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Lucas, S. M.; Liu, J.; and Pérez-Liébana, D. 2017. Efficient noisy optimisation with the multi-sample and sliding window compact genetic algorithms. In *Computational Intelligence (SSCI), 2017 IEEE Symposium Series on*. IEEE.

Lucas, S. M.; Liu, J.; and Perez-Liebana, D. 2018. The N-Tuple Bandit Evolutionary Algorithm for Game Agent Optimisation. In *2018 IEEE Congress on Evolutionary Computation (CEC)*.

Lucas, S. M. 2018. Game AI Research with Fast Planet Wars Variants. In *IEEE Conference on Computational Intelligence and Games*.

Perez-Liebana, D.; Liu, J.; Khalifa, A.; Gaina, R. D.; Togelius, J.; and Lucas, S. M. 2018. General Video Game AI: a Multi-Track Framework for Evaluating Agents, Games and Content Generation Algorithms. *arXiv preprint arXiv:1802.10363*.

Rapin, J., and Teytaud, O. 2018. Nevergrad - A gradient-free optimization platform. `https://GitHub.com/FacebookResearch/Nevergrad`.

Sironi, C. F., and Winands, M. H. M. 2017. On-line parameters tuning for Monte-Carlo tree search in general game playing. In *6th Workshop on Computer Games (CGW)*.

Sironi, C. F.; Liu, J.; Perez-Liebana, D.; Gaina, R. D.; Bravi, I.; Lucas, S. M.; and Winands, M. 2018. Self-adaptive mcts for general video game playing. In *European Conference on the Applications of Evolutionary Computation. Springer*.

Yannakakis, G. N., and Togelius, J. 2018. *Artificial Intelligence and Games*. Springer. `http://gameaibook.org`.