Chapter 4 - Frontiers of GVGAI Planning

Diego Perez-Liebana and Raluca D. Gaina

1 Introduction

Multiple studies have tackled the problem presented in the planning track of GVGAI. This chapter aims to present the state-of-the-art on GVGAI, describing the most successful approaches on this domain in Section 2. It is worth highlighting that these approaches are based on different variants of tree search methods, predominantly MCTS (see previous chapter), with the addition of several enhancements or heuristics. These agents submitted to the competition have shown the ability of playing multiple games minimizing the amount of game-dependent heuristics given, truly reflecting the spirit of GVGAI.

It is important to observe and analyze these methods because, despite being the most proficient ones, they are still far from an optimal performance on this problem. On average, these approaches achieve around 50% of victory rate on games played. From the study of the limitations of these methods one can build stronger algorithms that tackle this challenge better. Section 3 summarizes the open challenges that must be faced in order to make progress in this area.

One of the most promising options is to automatically extract information from the game played in an attempt to *understand* it. Different games may benefit from distinct approaches, and being able to identify when to use a determined algorithm or heuristic can provide a big leap forward. In a similar vein, being able to understand the situation of the agent in the game (i.e. is the agent on the right track to win?) can provide automatic guidelines as to when to change the playing strategy. Section 4 of this chapter describes our approach to predict game outcomes from live gameplay data [8]. This is meant to be another step towards a portfolio approach that improves the state of the art results in the planning challenge of general video game playing.

GVGAI planning is a hard problem, and we have reached a point where general agents can play decently some of the available games. The next step is maybe the hardest (but arguably the most interesting one): can we automatically analyze games and determine what needs to be done to win? How can we understand the goals of any game and how closely is the agent to achieve them? The answers to these questions may lead to a new generation of general video game playing agents.

2 State of the art in GVGAI Planning

This section presents three agents that have won several editions of the GVGAI competitions: OLETS, ToVo2 and YOLOBOT. We describe the methods implemented and the strengths and weaknesses of each approach.

2.1 OLETS (adrienctx)

Open Loop Expectimax Tree Search (OLETS) is an algorithm developed by Adrien Couëtoux for GVGAI that won the first edition of the single-player GVGAI competition. Furthermore, its adaptation to the 2-player planning track [4] was also successful at winning two editions of such track (see Chapter 2).

OLETS takes its inspiration from Hierarchical Open-Loop Optimistic Planning (HOLOP, [20], which is uses the Forward Model (FM) to sample actions in a similar way to MCTS, without ever storing the states of the tree in memory. Although this is not an optimal approach for non-deterministic domains, it works well in practice. One key difference with HOLOP is that OLETS does not use the default UCB policy for action selection but a new method called *Open Loop Expectimax* (OLE). OLE uses $r_M(n)$ instead of the average of rewards in the policy, which is the linear combination of two components: the empirical average reward computed from the simulations that went through the node n and the maximum r_M value among its children. Another key difference is that OLETS does not perform any rollouts. Equation 1 shows the OLE score, where $n_s(n)$ is the number of times the state s has been visited and $n_s(a)$ the number of times action a has been chosen from state s.

$$score = r_M(a) + \sqrt{\frac{ln(n_s(n))}{n_s(a)}} \tag{1}$$

Algorithm 1 describes OLETS in detail. OLETS tries to reward exploration of the level via a *taboo bias* added to the state value function. When new nodes are added to the tree, their state receives a penalty (of an order of magnitude smaller than 1) if it has been visited previously less than T steps ago. Values of $10 < T < 30$ has shown to provide good empirical results.

One of the weaknesses of OLETS is the absence of learning. It's a very simple and flexible method that gives the agent good reactive capabilities, mainly due to the open-loop search (in contrast to closed-loop that stores the states of the games in the node). However, it struggles in games where long computations are needed to solve a game that requires a deeper planning.

Algorithm 1 OLETS, from [13]. $n_s(n)$: number of simulations that passed through node n; $n_e(n)$: number of simulations ended in e; $R_e(n)$: cumulative reward from simulations ended in n; $C(n)$: set of children of n; $P(n)$: parent of n.

Input: s: current state of the game.
Input: T: time budget.
Output: *action*: to play by the agent in the game.

```
function OLETS($s$, $T$)
    $\mathcal{T}$ ←root initialize the tree
    while elapsed time $< T$ do
        RunSimulation($\mathcal{T}$, $s$)
    return action $= \arg\max_{a \in C(root)} n_s(a)$

function RunSimulation($\mathcal{T}$, $s_0$)
    $s \leftarrow s_0$ set initial state
    $n$ ←root($\mathcal{T}$) start by pointing at the root
    $Exit$ ←False
    while $\neg Final(s) \wedge \neg Exit$ do
        if $n$ has unexplored actions then
            $a$ ←Random unexplored action
            $s$ ←ForwardModel(s,a)
            $n$ ←NewNode(a,Score(s))
            $Exit$ ←True
        else
            $a \leftarrow \arg\max_{a \in C(n)}$OLE($n$,$a$) select a branch with OLE (Equation 1)
            $n \leftarrow a$
            $s$ ←ForwardModel(s,a)
    $n_e(n) \leftarrow n_e(n) + 1$
    $R_e(n) \leftarrow R_e(n) + Score(s)$
    while $\neg P(n) = \emptyset$ do
        $n_s(n) \leftarrow n_s(n) + 1$ update the tree
        $r_M(n) \leftarrow \frac{R_e(n)}{n_s(n)} + \frac{(1 - n_e(n))}{n_s(n)} \max_{c \in C(n)} r_M(c)$
        $n \leftarrow P(n)$
```

2.2 ToVo2

The ToVo2 controller is an agent developed by Tom Vodopivec and inspired in MCTS and Reinforcement Learning [16]. ToVo2 was the first winner of the 2-player GVGAI planning competition.

ToVo2 enhances MCTS by combining it with Sarsa-UCT(λ) [17] to combine the generalization of UCT with the learning capabilities of Sarsa. The algorithm is also an open-loop approach that computes state-action values using the FM during the selection and simulation phases of MCTS. Rewards are normalized to $[0, 1]$ to combine the UCB1 policy with Sarsa. Other parameters are an exploration rate $C = \sqrt{2}$, a reward discount rate $\gamma = 0.99$ and the eligibility trace decay rate $\lambda = 0.6$.

The simulation step computes the value of all states visited, instead of only observing the state found at the end of the rollout. This value is calculated as the difference in score between two consecutive game ticks. All visited nodes in an iteration are retained and 50% of knowledge is forgotten after each search due to the updated step-size parameter. The opponent is modeled as if it were to move uniformly at random (i.e. it is assumed to be part of the environment).

Two enhancements for the MCTS simulation phase augment the controller. The first one is weighted-random rollouts, which bias the action selection to promote exporation and visit the same state again less often. The second one is a dynamic rollout length, aimed at an initial exploration of the close vicinity of the avatar (starting with a depth of 5 moves from the root) to then increase the depth progressively (by 5 every 5 iterations, up to a maximum of 50) to search for more distant goals.

Similar to OLETS, ToVo2 struggles with games that see no rewards in the proximity of the player, but it is robust when dealing with events happening in its close horizon. In games where the exploration of the level is desirable, the weighted-random rollouts proved to be beneficial, but if offered no advantage in puzzle games where performing an exact and particular sequence of moves is required.

2.3 YOLOBOT

A submitted agent that is worth discussing is YOLOBOT, which won three editions of the GVGAI single-player planning track (outperforming OLETS in them). YOLOBOT was developed by Tobias Joppen, Miriam Moneke and Nils Schröder, and its full description can be found at [10].

YOLOBOT is a combination of two different methods: a heuristic-guided Best First Search and MCTS. The former is used in deterministic games, while the latter is employed in stochastic ones. The heuristic search attempts to find a sequence of moves that can be replicated exactly. At the start, the agent returns the NIL action until this sequence is found, a limit of game ticks is reached or the game is found to be stochastic. In the latter case, the agent switches to an enhanced MCTS to play the game. This algorithm is enhanced in several ways.

- Informed priors: YOLOBOT keeps and dynamically updates a knowledge base with predictions about events and movements within the game. This knowledge base is used in an heuristic to initialize the UCT values for non-visited nodes, biasing MCTS to expand first through those actions that seem to lead to more promising positions. These values are disregarded as soon as MCTS starts a simulation from them.

- Informed rollout policies: the same heuristic used to find promising positions is employed to bias action selection during the simulation phase of MCTS.
- Backtracking: when finding a losing terminal state, the algorithm backtracks one move and simulates up to four alternative actions. If one of these alternative moves leads to a non-losing state, the value of this one is backpropagated instead.
- Pruning of the search space: the FM is used to determine if the next state of s when applying action a is the same as the state reached when NIL is played. In this case, all actions a that meet this criteria are pruned. Other actions leading the avatar outside the level bounds and those that lead to a losing game state are also pruned.

YOLOBOT has achieved the highest win rate in GVGAI competitions since its inception. This agent works well in deterministic games (puzzles) where the solution is reachable in a short to medium-long sequence of actions, and it outperforms the other approaches when playing stochastic games. This, however, does not mean that YOLOBOT excels at all games. As shown in Section 2, YOLOBOT has achieved between 41.6% and 63.8% of victories in different game sets of the framework. There is still about 50% of the games where victory escapes even to one of the strongest controllers created for this challenge.

3 Current Problems in GVGAI Planning

Both planning tracks of GVGAI (single and 2-player) have received most attention since the framework was built and made public. As hinted in the previous section and shown in Chapter 2, the best approaches do not achieve a higher than 50% winning rate. Furthermore, many games are solved in very rare cases and the different MCTS and RHEA variants struggle to get more than 25% of victories in all (more than 100) games of the framework. Thus, one of the main challenges at the moment is to increase the win percentage across all games of the framework.

A recent survey [14] of methods for GVGAI describes many enhancements for algorithms that try to tackle this issue. In most cases, including the ones described in this book, the improvements do achieve to increase performance in a subset of the games tried, but there is normally another subset in which performance decreases or (in the best case) stays at the same level. The nature of GVGP makes this understandable, as it is hard to even think of approaches to work well for all games - but that does not mean this is not a problem to solve.

This recent survey shows, however, a few lines of work that could provide significant advances in the future. For instance, it seems clear that feature extraction from sprites work better if using a more sophisticated measure (not only a straightforward A*, but also other methods like potential fields [3]) than simple Euclidean distances. How to combine this more complex calculations with the real-time aspect of the games, especially in those games where a wise use of the budget is crucial, is still a matter for future investigation.

The wise use of the budget time is an important point for improvement. There is a proliferation of methods that try to use the states visited with the FM during the thinking time to learn about the game and extract features to bias further searches (as in [12] and [10]). In most cases, these approaches work well providing a marginal improvement in the overall case, but the features are still tailored to a group of games and lack generality. Put simply, when researchers design feature extractors, they are (naturally) influenced by the games they know and what is important in them, but some other games may require more complex or never-thought-before features. The design of an automatic and general feature extractor is one of the biggest challenges in GVGP.

Another interesting approach is to work on the action space to use more abstract moves. This can take the for of macro-actions (sequences of atomic actions treated as a whole [15]) or options (in MCTS [19], associated to goals). This approach makes the action space coarser (reduces the action space across several consecutive turns) and maximizes the use of the budget times: once a macro-action starts its execution (which takes T time steps to finish), the controller can plan for the next macro-action, counting on $T - 1$ time steps to complete that search. This can be an interesting approach for games that require long-term planning, a subset of games that has shown to be hard for the current approaches. Results of using these action abstractions show again that they help in some games, but not in others. They also suggest that some games benefit from different sets of lengths of macro-actions, while other games are better played with others.

It is reasonable to think that, given that certain algorithms perform better at some games than others, and some games are played better by different methods, an approach that tries to automatically determine what is the right algorithm for the right game should be of great help. In fact, the already discuss YOLOBOT agent takes a first stab at this, by distinguishing between deterministic and stochastic games. Game classification [11][1] and the use of hybrids or hyper-heuristic methods are in fact an interesting area for research. The objective would be to build a classifier that dynamically determines which is the best algorithm to use in the current game and then switch to it. Some attempts have been made to classify games using extracted features, although the latest results seem to indicate that these classifications (and the algorithms used) are not strong enough to perform well across many games.

One interesting direction that we start exploring in the next chapter is the use of more general features, focused on how the agent *experiences* the game rather than features that are intrinsic to it (and therefore biased). The next study puts the first brick in a system designed to switch between algorithms in game based only on agent game-play features. In particular, the next section describes how a win predictor can be built to determine the most likely outcome of the game based on agent experience.

4 General Win Prediction in GVGAI

The objective of this work is to build an outcome predictor based only on measurements from the agent's experience while playing any GVGAI game. We purposefully left game-related features (like the presence of NPCs or resources) outside this study, so the insights can be transferred to other frameworks or particular games without much change. The most important requirements are that the game counts on a Forward Model (FM), a game score and game end states with a win/loss outcome.

4.1 Game Playing Agents and Features

The first step for building these predictors is to gather the data required to train the predicting models. This data will be retrieved from agents playing GVGAI games. The algorithms used will be as follows.

Random Search (RS) The RS agent samples action sequences at random of length L until the allocated budget runs out. Then, the first action of the best sequence is played in the game. Sequences are evaluated using the FM to apply all actions from the current game state and assigning a value to the final state following Equation 2. H^+ is a large positive integer (and H^- is a large negative number).

$$f = score \; + \; \begin{cases} H^+, \; if \; loss \\ H^-, \; if \; win \end{cases} \tag{2}$$

Three configurations for RS are used in this study, according to the value of L: 10, 30 and 90.

Rolling Horizon Evolutionary Algorithm (RHEA) The RHEA agent (explained in Chapter 2 is used for this study, using some of the improvements that have shown to work well in previous studies. The selected algorithm configurations are:

- Vanilla RHEA [5], the default configuration of the algorithm.
- EA-MCTS [6], in which the original population is seeded by MCTS.
- EA-Shift [7], in which RHEA is enhanced using the Shift Buffer and Monte Carlo rollouts at the end of the individuals.
- EA-All, which combines EA-Shift with EA-MCTS, for completeness.

For these four variants, two different parameters sets where used: P=2, L=8 and P=10, L=14, where L is individual length and P population size. Final states are evaluated using Equation 2.

Monte Carlo Tree Search (MCTS) The vanilla MCTS algorithm is used, using again Equation 2 to evaluate states reached at the end of the rollouts. 3 parameter sets were used for MCTS: W=2, L=8; W=10, L=10 and W=10, L=14. W is the number of MCTS iterations and L the depth of the rollouts from the root state.

All these 14 algorithms were run in the 100 public games of the GVGAI framework, using the 5 levels available per game and 20 repetitions per level. All algorithms counted on the same budget per game tick to make a decision: 900 calls to the FM's advance function. Table 1 summarizes the results obtained by these methods in the 100 games tested.

Each one of these runs produced two log files with agent and game state information at every game state. Regarding game, the score at each step and the final game result (win/loss) are saved. Regarding the agent, we logged the action played at each game step and the set of features described as follows.

- ϕ_1 **Current game score**.
- ϕ_2 **Convergence**: The iteration number when the algorithm found the final solution recommended and did not change again until the end of the evolution, during one game step. A low value indicates quick and almost random decisions.
- ϕ_3 **Positive rewards**: The count of positive scoring events.
- ϕ_4 **Negative rewards**: The count of negative scoring events.
- ϕ_5 **Success**: The slope of a line over all the win counts. This count reflects the number of states which ended in a win at any point during search. A high value shows an increase in discovery of winning states as the game progresses.

#	**Algorithm**	**Victory Rate (Standard Error)**
1	**10-14-EA-Shift**	26.02 % (2.11)
2	**2-8-EA-Shift**	24.54 % (2.00)
3	**10-RS**	24.33 % (2.13)
4	**14-MCTS**	24.29 % (1.74)
5	**10-MCTS**	24.01 % (1.65)
6	**10-14-EA-MCTS**	23.99 % (1.80)
7	**2-8-EA-MCTS**	23.98 % (1.73)
8	**2-8-EA-All**	23.95 % (1.98)
9	**8-MCTS**	23.42 % (1.61)
10	**10-14-RHEA**	23.23 % (2.08)
11	**10-14-EA-All**	22.66 % (2.02)
12	**30-RS**	22.49 % (2.02)
13	**2-8-RHEA**	18.33 % (1.77)
14	**90-RS**	16.31 % (1.67)

Table 1: Victory rate (and standard error) of all methods used in this study across 100 GVGAI games. Type and configuration (rollout length L if one value, population size P and roll-out length L if two values) are reported.

- ϕ_6 **Danger**: The slope of a line over all the loss counts. This count grows for every end game loss found during search. A high value indicates that the number of losing states increases as the game progresses.
- ϕ_7 **Improvement**: Given the best fitness values seen since the beginning of the game, improvement is the slope of a this increment over game tick. A high value indicates that best fitness values increase as the game progresses.
- ϕ_8 **Decisiveness**: This is the Shannon Entropy (SE, see Equation 3) over how many times each action was recommended. In all cases where the feature is calculated as SE, a high value suggests actions of similar value; the opposite shows some of these actions to be recommended more often.
- ϕ_9 **Options exploration**: SE over the number of times each of the possible actions was explored. In this case, this reflects how many times this action was the first move of a solution at any time during search. A low value shows an imbalance in actions explored while a high value means that all actions are explored approximately the same as first moves during search.
- ϕ_{10} **Fitness distribution**: SE over fitness per action.
- ϕ_{11} **Success distribution**: SE over win count per action.
- ϕ_{12} **Danger distribution**: SE over loss count per action.

$$H(X) = -\sum_{i=0}^{N-1} p_i log_2 p_i \tag{3}$$

Features ϕ_2, ϕ_8, ϕ_9, ϕ_{10}, ϕ_{11} and ϕ_{12} compute averages from the beginning of the game up until the current tick t. Features ϕ_5, ϕ_6, ϕ_{11} and ϕ_{12} rely on the FM. Data set and processing scripts have been made publicly available[1].

Additionally, features are divided into 3 different game *phases*: early (first 30% ticks of the games), late (ticks from the last 70% of the game) and mid-game (ticks between 30% – 70% of the game). These divisions were used to train different phase models: early, mid and late game classifiers.

Feature correlation The division in different game stages corresponds to the belief that different events generally occur at the beginning and end of the game. Therefore, having models trained in different game phases may produce interesting results. Figure 1 shows the correlation between features using the Pearson correlation coefficient. This compares the the early (left) and late (right) game features showing small (but existing) differences. For instance, there are higher correlations between the features located in the bottom right corner in the late game than in the early game.

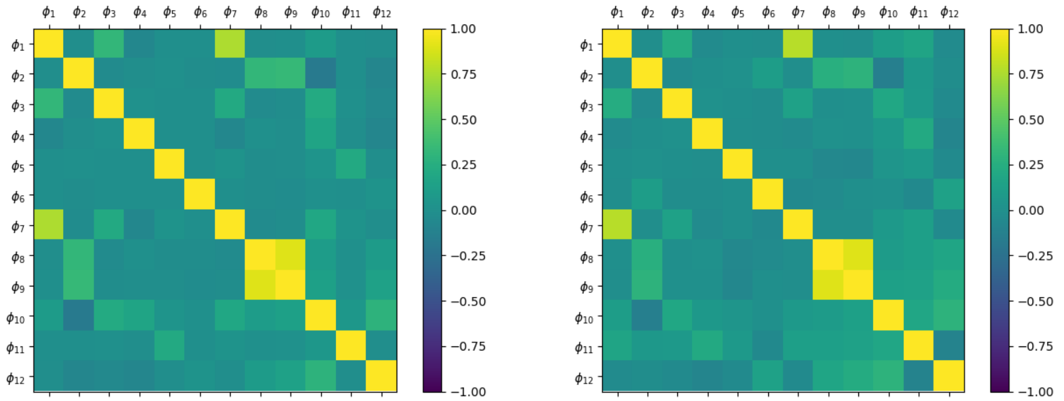

Fig. 1: Feature correlation early game (left, 0-30% of all games) and late game (right, 70-100% of all games)

[1] https://github.com/rdgain/ExperimentData/tree/GeneralWinPred-CIG-18. The final data is split over 281,000 files. It took approximately 2.5 hours to generate this database from raw data files, using a Dell Windows 10 PC, 3.4 GHz, Intel Core i7, 16GB RAM, 4 cores.

The late game phase also shows an interesting and positive correlation between convergence (ϕ_2) and danger (ϕ_6). This may suggest that agents take more time to settle on their final decisions when more possible losses are found within the search. A positive correlation that is less strong in the late game is that between fitness improvement and fitness distribution over the actions, implying that when one action is deemed significantly better than the rest, it is unlikely for the fitness to improve further, possibly due to the other actions not being explored further. Finally, a persistent negative correlation exists between convergence and fitness distribution. This seems to indicate that an agent, when finding an action that is deemed prominently better than the rest, it is not likely to change its decision.

4.2 Predictive models

For all models trained for this study the data was randomly split in training (80 of the data) and test sets (20). Several classifier models are built using the features described as input and a win/loss as a label and prediction. Model prediction quality is reported in this section according to precision, recall and F1 score (Equations 4, 5 and 6, respectively):

$$precision = \frac{TP}{TP + FP} \tag{4}$$

$$recall = \frac{TP}{TP + FN} \tag{5}$$

$$F1 = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{6}$$

TP stands for true positives (correctly predicted a win), FP stands for false positives (incorrectly predicted a win) and FN stands for false negatives (incorrectly predicted a loss). F1 score is the main indicator reported due to the low overall performance of the general agents (close to 25% win rate in all games) and the high variety of the GVGAI games.

Baseline Model In order to determine if the models trained are better than a simple expert rule system, a baseline model was built. In most games (GVGAI and other arcades), gaining score through the game is typically a good indicator of progression, and as such it normally leads to a victory. Equation 7 is a simple model that compares the count of positive and negative score events.

$$\hat{y} = \begin{cases} win & if\ \phi_3 > \phi_4 \\ lose & otherwise \end{cases} \tag{7}$$

The performance of this classifier on the test set can be seen in Table 2. It can be observed that it achieves an an F1-Score of 0.59, despite having a high precision (0.70). This model is referred to as R_g in the rest of this chapter.

	Precision	**Recall**	**F1-Score**	**Support**
Loss	0.83	0.52	0.64	20500
Win	0.35	0.70	0.46	7500
Avg / Total	**0.70**	**0.57**	**0.59**	**28000**

Table 2: Global rule based classifier report. Global model tested on all game ticks of all instances in the test set.

Classifier selection - global model In this work, we trained and tested seven classifiers as a proof of concept for the win prediction task. These classifiers are K-Nearest Neighbors (5 neighbours), Decision Tree (5 max depth), Random Forest (5 max depth, 10 estimators), Multi-layer Perceptron (learning rate 1), AdaBoost-SAMME [21], Naive Bayes and Dummy (a simple rule system, also used as another baseline). All classifiers use the implementation in the Scikit-Learn Python 2.7.14 library [2]. The parameters not specified above are set to their respective default values in the Scikit-Learn library.

Performance is assessed using cross-validation (10 folds). The classifiers obtained, following the same order as above, an accuracy of 0.95, 1.00, 0.98, 0.96, 1.00, 0.95 and 0.66 during evaluation. AdaBoost and Decision Tree achieved very high accuracy values in validation and test (see Table 3). The rest of the experiments use AdaBoost as the main classifier (with no particular reason over Decision Trees).

Table 4 shows the importance of teach feature according to AdaBoost. Game score seems to be, unsurprisingly, the most important feature to distinguish wins and losses. This is closely followed by the number of wins seen by the agent, the improvement of the fitness and the measurement of danger. Decisiveness of the agent seems to have no impact in deciding the outcome of the game, according to the model trained.

	Precision	**Recall**	**F1-Score**	**Support**
Loss	1.00	0.99	0.99	20500
Win	0.97	0.99	0.98	7500
Avg / Total	**0.99**	**0.99**	**0.99**	**28000**

Table 3: AdaBoost tested on all game ticks of all instances in the test set.

ϕ_1	0.24	ϕ_2	0.04	ϕ_3	0.08	ϕ_4	0.06
ϕ_5	0.2	ϕ_6	0.1	ϕ_7	0.12	ϕ_8	0
ϕ_9	0.06	ϕ_{10}	0.02	ϕ_{11}	0.02	ϕ_{12}	0.06

Table 4: Importance of features as extracted from the global model.

Model training Predictions were made at three different levels during the game: early game (first 30% of the game ticks), mid game ($30 - 70\%$) and late game (last 30%). The models are trained using the features captured in the game ticks corresponding to each interval in the game. These three models are referred to as E_g, M_g and L_g, respectively, in the rest of this section.

Trained models were tested by checking their performance on the 20 test games (also considering the game tick intervals). Training with 10-fold cross-validation provided 0.80, 0.82 and 0.99 as results for the early, mid and late game predictions, respectively. Test accuracies are 0.73, 0.80 and 0.99, with F1-Scores of 0.70, 0.80 and 0.99, respectively.

Live play results Our next step was to simulate how the trained models predict the game outcome *live* (when an agent is playing). For this, we simulated play by extracting the features logged by agents in log files for a range of game ticks ($T = \{100 \cdot a : \forall\ a \in [1, 20] : a \in N\}$), all from the beginning of the game until the current tick tested $t \in T$. We took games played by the 14 algorithms presented above on 20 test games, playing 20 times on each one of their 5 levels. Each model was asked to predict the game outcome every 100 ticks.

Figures 2 and 3 show the results obtained from this testing. The baseline rule-based model achieves a high performance in some games, showing to be better than the trained predictive models (i.e. see *Aliens*, *Defem*, *Chopper* and *Eggomania*). These games have plenty of scoring events, thus it is not surprising that the simple logic of R_g works well in these cases. However, there are other games in which the trained models perform much better predictions than the baseline (see *Ghost Buster*, *Colour Escape* or *Frogs*), where the outcome is not significantly correlated with game score and the simple rule-based prediction is not accurate.

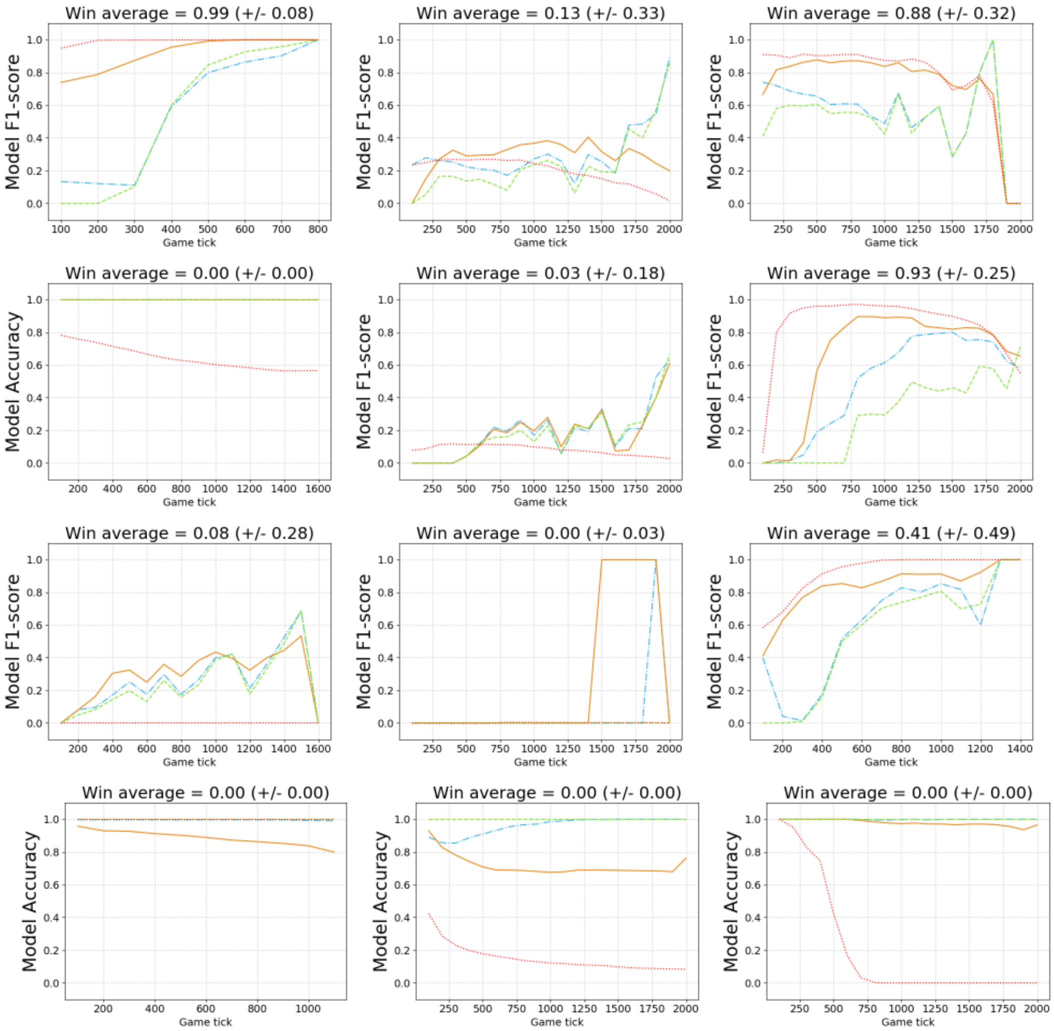

Fig. 2: Model F1-scores for each game in the test set, averaged over up to 1400 runs, 14 agents, 100 runs per game. Game ticks are displayed on the X axis, maximum 2000 game ticks. Three different predictor models trained on early, mid and late game data features, as well as the baseline rule-based predictor. If F1-scores were 0 for all models, accuracy is plotted instead. Additionally, win average is reported for each game. Games from top to down, left to right: *Aliens*, *Boulderdash*, *Butterflies*, *Caky Baky*, *Chase*, *Chopper*, *Colour Escape*, *Cops*, *Defem*, *Deflection*, *DigDug* and *Donkeykong*.

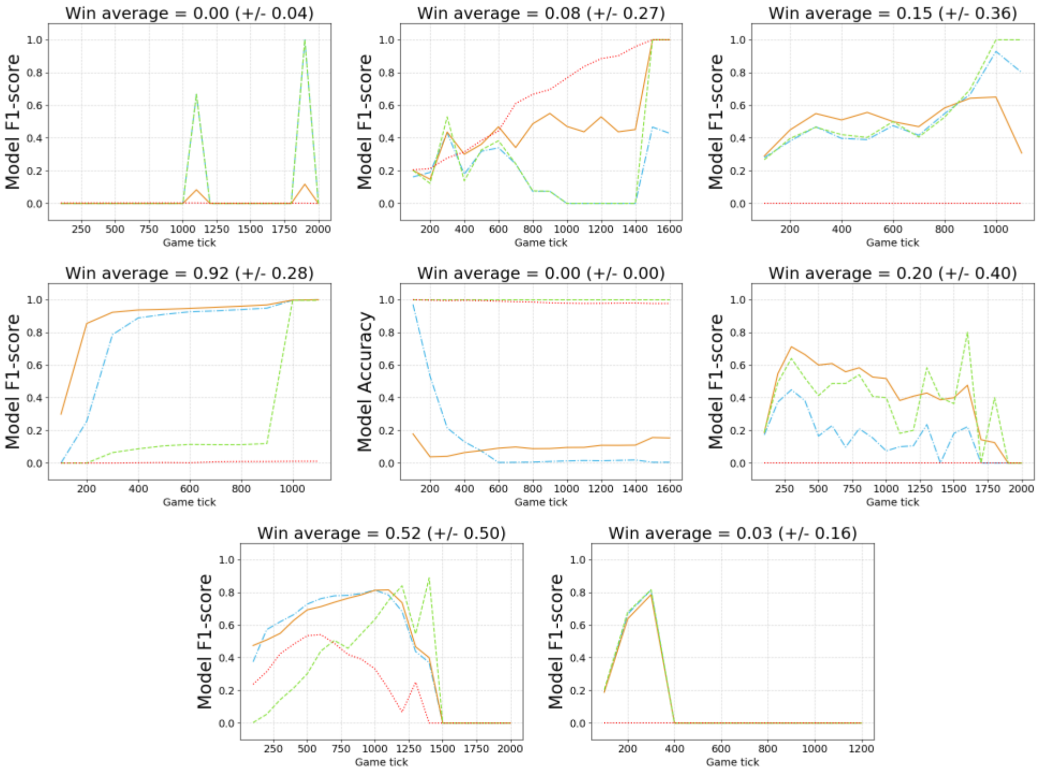

Fig. 3: Model F1-scores for each game in the test set, averaged over up to 1400 runs, 14 agents, 100 runs per game. Game ticks are displayed on the X axis, maximum 2000 game ticks. Three different predictor models trained on early, mid and late game data features, as well as the baseline rule-based predictor. If F1-scores were 0 for all models, accuracy is plotted instead. Additionally, win average is reported for each game. Games from top to down, left to right: *Dungeon*, *Eggomania*, *Escape*, *Factory Manager*, *Fireman*, *Frogs*, *Ghost Buster* and *Hungry Birds*.

It is worthwhile mentioning that the trained models do not follow the expected curves. One could expect E_g performing better in the early game, to then decrease its accuracy when the game progresses. M_g could show high performance in the middle game and L_g offering good predictions only on the end game. However, the early game predictor has a worst performance compared to the rest (this could be explained by the lack of information available for this model). The late game model is very accurate in games with very low win rate (in *Fireman*, for instance, where E_g and M_g are predicting wins, yet the overall win rate remains at 0% for this game).

A high F1-score index indicate that the predictors are able to judge correctly both wins and losses. It is therefore interesting to pay attention to those games, like *Defem* and *Ghost Buster*, with close to 50% win rate. In these cases, F1-scores over 0.8 are achieved when only half the game has passed. The middle model M_g provides very good results in this situations, becoming the best predictor to use in this case and possibly the best one to use.

It is remarkable to see that the model is able to predict, half-way through the game (and sometimes just after only a fourth of the game has been played), the outcome of the game, even if games are won or lost with the probability as a coin flip. These models are general: they have been trained in different games without relaying in game-dependent features - just agent experience measurements.

Therefore, there is a great scope of using these predictors as part of a hyper-heuristic system. Some of the algorithm tested in this study *do* win at these 50% win rate games like *Defem* or *Ghost Buster* and finding a way to use the appropriate method for each game would boost performance in GVGAI. Such system would need to count on an accurate win prediction model (to know if switching to a different method is required) and a second model that determines which is the best method given the features observe (to know what to change to).

Table 5 summarises the F1-scores of the three models on the different game phases identified over all games. Results shown in this table indicate that the rule-based model provides a consistent performance in all game phases. It is better than the others in the early phase (F1-score of 0.42). However, in the middle and late game phases, M_g is significantly better than all the others (F1-scores of 0.57 and 0.71, respectively). Over all games and phases, the middle game M_g model is the best one with an average F1-score of 0.53. M_g is the strongest model in the individual mid and late phases, only overcome by the simple rule predictor (which incorporates human knowledge as it considers that gaining score leads to a victory) in the early game phase.

	Early-P	**Mid-P**	**Late-P**	**Total-M**
E_g	0.22 (0.72)	0.42 (0.74)	0.49 (0.76)	0.38 (0.74)
M_g	0.29 (0.72)	**0.57 (0.79)**	**0.71 (0.83)**	**0.53 (0.78)**
L_g	0.01 (0.73)	0.05 (0.74)	0.22 (0.76)	0.09 (0.74)
R_g	**0.42 (0.67)**	0.47 (0.61)	0.46 (0.58)	0.45 (0.62)
Total-P	0.24 (0.71)	0.38 (0.72)	**0.47 (0.73)**	

Table 5: F1-Scores each model per game phase over all games, accuracy in brackets. Each column is a game phase, each row is a model. Highlighted in bold is the best model on each game phase, as well as overall best phase and model.

Finally, in order to test the robustness of the predictions, we play-tested the test games with MCTS while using the models trained with RHEA. The agent experience features extracted from an MCTS agent are fed into the prediction models trained with a different algorithm. Figure 4 shows a comparison between this testing and the previous one. On the left, the model being used by an MCTS agent. On the right, when played with the RHEA and RS variants. As can be seen, all models behave similarly in the different stages of the game and are able to accurately predict the game outcome half-way through it.

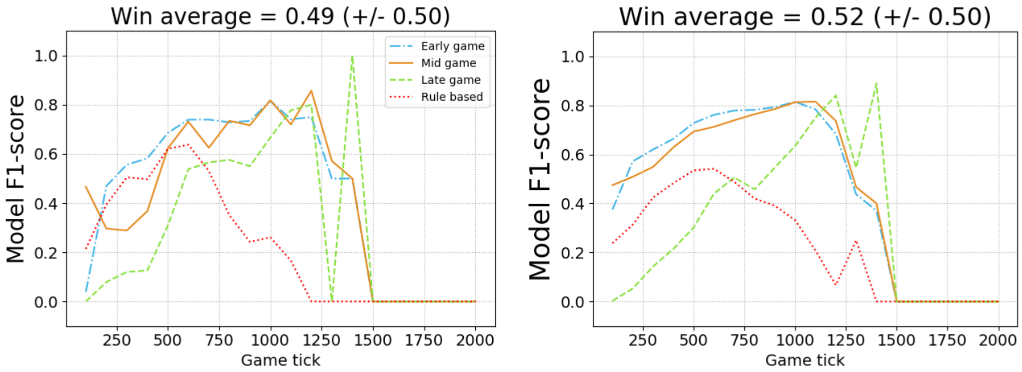

Fig. 4: F1-scores achieved in the game *Ghost Buster*, trained (in 80 training games) with RHEA and RS and tested (on 20 test games) with MCTS (left) and RHEA (right).

Next Steps As mentioned before, a logical next step would be to build a hyper-heuristic GVGAI agent that can switch between algorithms, in light of the predictions, when playing any game. Identifying which is the algorithm to switch to can be decomposed in two sub-tasks: one of them signalling which feature measurements need to change and the other identifying which agent can deliver the desired new behaviour.

Regarding the first task, it is possible to analyze how the different features influence each task. Figure 5 shows an example of the prediction given by the three models at game tick 300 in *Frogs* (level 0, played by 2-8-RHEA). As can be seen, different features indicators are highlighted for each model, signaling which features are *responsible* for the win or loss predictions. It is possible that a hyper-heuristic method that makes uses of this analysis could determine the reasons for the predictions of the models.

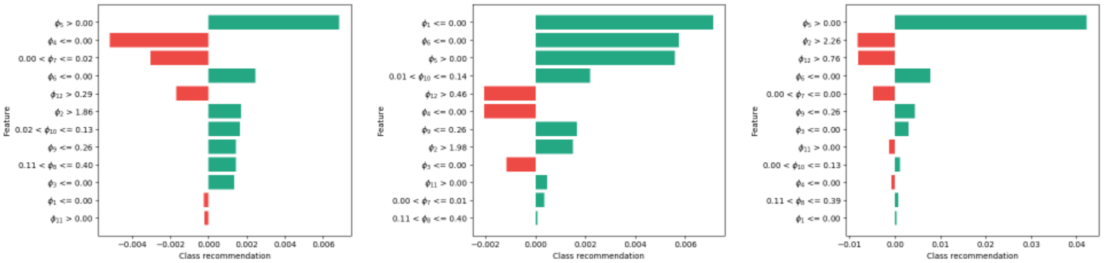

Fig. 5: Class predictions by features (LIME system[2]). Red signifies the model feature recommends a loss, green a win. The probability of class being selected based on individual feature recommendation is plotted on the X-axis.

Finally, the model could be enhanced with deep variants and more features, such as empowerment [9], spatial entropy or characterization of agent surroundings [18].

References

1. P. Bontrager, A. Khalifa, A. Mendes, and J. Togelius, "Matching Games and Algorithms for General Video Game Playing," in *Twelfth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2016.
2. L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux, "API design for machine learning software: experiences from the scikit-learn project," in *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, 2013, pp. 108–122.
3. C. Y. Chu, T. Harada, and R. Thawonmas, "Biasing Monte-Carlo Rollouts with Potential Field in General Video Game Playing," , pp. 1–6, 2015.
4. R. D. Gaina, A. Couëtoux, D. J. Soemers, M. H. Winands, T. Vodopivec, F. Kirchgessner, J. Liu, S. M. Lucas, and D. Perez-Liebana, "The 2016 Two-Player GVGAI Competition," *IEEE Transactions on Computational Intelligence and AI in Games*, 2017.
5. R. D. Gaina, J. Liu, S. M. Lucas, and D. Pérez-Liébana, "Analysis of Vanilla Rolling Horizon Evolution Parameters in General Video Game Playing," in *European Conference on the Applications of Evolutionary Computation*. Springer, 2017, pp. 418–434.
6. R. D. Gaina, S. M. Lucas, and D. Pérez-Liébana, "Population Seeding Techniques for Rolling Horizon Evolution in General Video Game Playing," in *Conference on Evolutionary Computation*. IEEE, 2017.
7. ——, "Rolling Horizon Evolution Enhancements in General Video Game Playing," in *2017 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 2017.
8. R. D. Gaina, S. M. Lucas, and D. Perez-Liebana, "General Win Prediction from Agent Experience," in *Computational Intelligence and Games (CIG), IEEE Conference on*, 2018.
9. C. Guckelsberger, C. Salge, J. Gow, and P. Cairns, "Predicting player experience without the player.: An exploratory study," in *Proceedings of the Annual Symposium on Computer-Human Interaction in Play*, ser. CHI PLAY '17. New York, NY, USA: ACM, 2017, pp. 305–315.

[2] `https://github.com/marcotcr/lime`

10. T. Joppen, M. U. Moneke, N. Schröder, C. Wirth, and J. Fürnkranz, “Informed Hybrid Game Tree Search for General Video Game Playing,” *IEEE Transactions on Games*, vol. 10, no. 1, pp. 78–90, 2018.
11. M. J. Nelson, “Investigating Vanilla MCTS Scaling on the GVG-AI Game Corpus,” in *2016 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 2016, pp. 402–408.
12. D. Perez, S. Samothrakis, and S. Lucas, “Knowledge-based Fast Evolutionary MCTS for General Video Game Playing,” in *Conference on Computational Intelligence and Games*. IEEE, 2014, pp. 1–8.
13. D. Perez, S. Samothrakis, J. Togelius, T. Schaul, S. Lucas, A. Couëtoux, J. Lee, C.-U. Lim, and T. Thompson, “The 2014 General Video Game Playing Competition,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 8, pp. 229–243, 2015.
14. D. Perez-Liebana, J. Liu, A. Khalifa, R. D. Gaina, J. Togelius, and S. M. Lucas, “General video game ai: a multi-track framework for evaluating agents, games and content generation algorithms,” *arXiv preprint arXiv:1802.10363*, 2018.
15. D. Pérez-Liébana, M. Stephenson, R. D. Gaina, J. Renz, and S. M. Lucas, “Introducing Real World Physics and Macro-Actions to General Video Game AI,” in *Conference on Computational Intelligence and Games (CIG)*. IEEE, 2017.
16. R. S. Sutton and A. G. Barto, *Reinforcement Learning : An Introduction*. MIT Press, 1998.
17. T. Vodopivec, S. Samothrakis, and B. Ster, “On monte carlo tree search and reinforcement learning,” *Journal of Artificial Intelligence Research*, vol. 60, pp. 881–936, 2017.
18. V. Volz, D. Ashlock, S. Colton, S. Dahlskog, J. Liu, S. M. Lucas, D. P. Liebana, and T. Thompson, “Gameplay Evaluation Measures,” in *Articial and Computational Intelligence in Games: AI-Driven Game Design (Dagstuhl Seminar 17471)*, E. André, M. Cook, M. Preuß, and P. Spronck, Eds. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, pp. 36–39.
19. M. d. Waard, D. M. Roijers, and S. C. Bakkes, “Monte Carlo Tree Search with Options for General Video Game Playing,” in *2016 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 2016, pp. 47–54.
20. A. Weinstein and M. L. Littman, “Bandit-Based Planning and Learning in Continuous-Action Markov Decision Processes,” in *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling, ICAPS, Brazil*, 2012.
21. J. Zhu, S. Rosset, H. Zou, and T. Hastie, “Multi-class AdaBoost,” *Ann Arbor*, vol. 1001, no. 48109, p. 1612, 2006.