Imitative Learning for Real-Time Strategy Games

Quentin Gemine, Firas Safadi, Raphaël Fonteneau and Damien Ernst

***Abstract*—Over the past decades, video games have become increasingly popular and complex. Virtual worlds have gone a long way since the first arcades and so have the artificial intelligence (AI) techniques used to control agents in these growing environments. Tasks such as world exploration, constrained pathfinding or team tactics and coordination just to name a few are now default requirements for contemporary video games. However, despite its recent advances, video game AI still lacks the ability to learn. In this paper, we attempt to break the barrier between video game AI and machine learning and propose a generic method allowing real-time strategy (RTS) agents to learn production strategies from a set of recorded games using supervised learning. We test this imitative learning approach on the popular RTS title StarCraft II® and successfully teach a Terran agent facing a Protoss opponent new production strategies.**

I. Introduction

Video games started emerging roughly 40 years ago. Their purpose is to bring entertainment to the people by immersing them in virtual worlds. The rules governing a virtual world and dictating how players can interact with objects or with one another are referred to as game mechanics. The first video games were very simple: small 2-dimensional discrete space, less than a dozen mechanics and one or two players at most. Today, video games feature large 3-dimensional spaces, hundreds of mechanics and allow numerous players and agents to play together. Among the wide variety of genres, real-time strategy (RTS), portrayed by games like Dune II (Westwood Studies, 1992), Warcraft (Blizzard Entertainment, 1994), Command & Conquer (Westwood Studios, 1995) or StarCraft (Blizzard Entertainment, 1998), provides one of the most complex environments overall. The multitude of tasks and objects involved as well as the highly dynamic environment result in extremely large and diverging state and action spaces. This renders the design of autonomous agents difficult. Currently, most approaches largely rely on generic triggers. Generic triggers aim at catching general situations such as being under attack with no consideration to the details of the attack (i.e., location, number of enemies, ...). These methods are easy to implement and allow agents to adopt a robust albeit non-optimal behavior in the sense that agents will not fall into a state for which no trigger is activated, or in other words a state where no action is taken. Unfortunately, this type of agent will often discard crucial context elements and fail to display the natural and intuitive behavior we may expect. Additionally, while players get more familiar with the game mechanics and improve their skills and devise new strategies, agents do not change and eventually become obsolete. This evolutionary requirement is critical for performance in RTS games where the pool of possible strategies is so large that it is impossible to estimate optimal behavior at the time of development. Although it is common to increase difficulty by granting agents an unfair advantage, this approach seldom results in entertainment and either fails to deliver the sought-after challenge or ultimately leads to player frustration.

Because the various facets of the RTS genre constitute very distinct problems, several learning technologies would be required to grant agents the ability to learn on all aspects of the game. In this work, we focus on the production problem. Namely, we deal with how an agent takes production-related decisions such as building a structure or researching a technology. We propose a generic method to teach an agent production strategies from a set of recorded games using supervised learning. We chose StarCraft II as our testing environment. Today, StarCraft II, Blizzard Entertainment's successor to genre patriarch StarCraft, is one of the top selling RTS games. Featuring a full-fledged game editor, it is the ideal platform to assess this new breed of learning agents. Our approach is validated on the particular scenario of a one-on-one, Terran versus Protoss matchup type. The created agent architecture comprises both a dynamically learned production model based on multiple neural networks as well as a simple scripted combat handler.

The paper is structured as follows. Section 2 briefly covers some related work. Section 3 details the core mechanics characterizing the RTS genre. Section 4 and 5 present the learning problem and the proposed solution, respectively. Section 6 discusses experimental results and, finally, Section 7 concludes and highlights future lines of work.

II. Related Work

Lately, video games have attracted substantial research work, be it for the purpose of developing new technologies to boost entertainment and replay value or simply because modern video games have become an alternate, low-cost yet rich environment for assessing machine learning algorithms.

Roughly, we could distinguish 2 goals in video game AI research. Some work aims at creating agents with properties that make them more fun to play with such as human-like behavior [1, 2]. Competitions like BotPrize or the Turing test track of the Mario AI Championship focus on this goal. It is usually attempted on games for which agents capable of challenging skilled human players already exist and is necessary because, often, agents manage to rival human players due to unfair advantages: instant reaction time, perfect aim, etc. These features increase performance at the cost of frustrating human opponents. For more complicated games, agents stand no chance against skilled human players and improving their performance takes priority. Hence, performance similar to

what humans can achieve can be seen as a prerequisite to entertainment. Indeed, we believe that facing a too weak or too strong opponent is not usually entertaining. This concept is illustrated in Figure 1. In either case, video game AI research advances towards the ultimate goal of mimicking human intelligence. It was in fact suggested that human-level AI can be pursued directly in these new virtual environments [3].

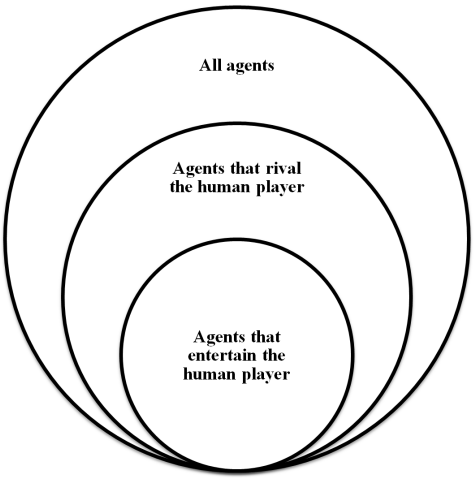


Fig. 1. Agent set structure for a video game

The problem of human-like agent behavior has been tackled in first-person shooter (FPS) games, most notably the popular and now open-source game Quake II, using imitative learning. Using clustering by vector quantization to organize recorded game data and several neural networks, more natural movement behavior as well as switching between movement and aim was achieved in Quake II [4]. Human-like behavior was also approached using dedicated neural networks for handling weapon switching, aiming and firing [5]. Further work discussed the possibility of learning from humans at all levels of the game, including strategy, tactics and reactions [6].

While human-like agent behavior was being pursued, others were more concerned with performance issues in genres like real-time strategy (RTS) where the action space is too large to be thoroughly exploited by generic triggers. Classifiers based on neural networks, Bayesian networks and action trees assisted by quality threshold clustering were successfully used to predict enemy strategies in StarCraft [7]. Case-based reasoning has also been employed to identify strategic situations in Wargus, an open-source Warcraft II clone [8, 9, 10]. Other works resorted to data mining and evolutionary methods for strategy planning and generation [11, 12]. Non-learning agents were also proposed [13]. By clearly identifying and organizing tasks, architectures allowing incremental learning integration at different levels were developed [14].

Although several different learning algorithms were applied in RTS environments, few were actually used to dictate agent behavior directly. In this paper, we use imitative learning to teach a StarCraft II agent to autonomously pass production orders. The created agent building, unit and technology production is entirely governed by the learning algorithm and does not involve any scripting.

III. Real-Time Strategy

In a typical RTS game, players confront each other on a specific map. The map is essentially defined by a combination of terrain configuration and resource fields. Once the game starts, players must simultaneously and continuously acquire resources and build units in order to destroy their opponents. Depending on the technologies they choose to develop, players gain access to different unit types each with specific attributes and abilities. Because units can be very effective against others based on their type, players have to constantly monitor their opponents and determine the combination of units which can best counter the enemy's composition. This reconnaissance task is referred to as scouting and is necessary because of the "fog of war", which denies visibility to players over areas where they have no units deployed.

Often, several races are available for the players to choose from. Each race possesses its own units and technologies and is characterized by a unique play style. This further adds to the richness of the environment and multiplies mechanics. For example, in StarCraft II players can choose between the Terrans, masters of survivability, the Zerg, an alien race with massive swarms, or the Protoss, a psychically advanced humanoid species.

Clearly, players are constantly faced with a multitude of decisions to make. They must manage economy, production, reconnaissance and combat all at the same time. They must decide whether the current income is sufficient or new resource fields should be claimed, they must continuously gather information on the enemy and produce units and develop technologies that best match their strategies. Additionally, they must swiftly and efficiently handle units in combat.

When more than two players are involved, new diplomacy mechanics are introduced. Players may form and break alliances as they see fit. Allies have the ability to share resources and even control over units, bringing additional management elements to the game.

Finally, modern RTS games take the complexity a step further by mixing in role-playing game (RPG) mechanics. Warcraft III, a RTS title also developed by Blizzard Entertainment™, implements this concept. Besides regular unit types, heroes can be produced. Heroes are similar to RPG characters in that they can gain experience points by killing critters or enemy units to level up. Leveling up improves their base attributes and grants them skill points which can be used to upgrade their special abilities.

With hundreds of units to control and dozens of different unit types and special abilities, it becomes clear that the RTS genre features one of the most complex environments overall.

IV. Problem Statement

The problem of learning production strategies in a RTS game can be formalized as follows.

Consider a fixed player u. A world vector $\boldsymbol{w} \in \mathcal{W}$ is a vector describing the entire world at a particular time in the game. An observation vector $\boldsymbol{o} \in \mathcal{O}$ is the projection of $\boldsymbol{w}$ over an observation space $\mathcal{O}$ describing the part of the world perceivable by player u. We define a state vector $\boldsymbol{s} \in \mathcal{S}$ as the projection of $\boldsymbol{o}$ over a space $\mathcal{S}$ by selecting variables deemed relevant to the task of learning production strategies. Let $n \in \mathbb{N}$ be the number of variables chosen to describe the state. We have:

$$\boldsymbol{s} = (s_1, s_2, ..., s_n), \forall i \in \{1, ..., n\} : s_i \in \mathbb{R}$$

Several components of $\boldsymbol{s}$ are variables that can be directly influenced by production orders. Those are the variables that describe the number of buildings of each type available or planned, the cumulative number of units of each type produced or planned and whether each technology is researched or planned. If a technology is researched or planned, the corresponding variable is equal to 1, otherwise, it is equal to 0. Let m be the number of these variables and let s_{p_1}, s_{p_2}, ..., s_{p_m} be the components of $\boldsymbol{s}$ that correspond to these variables.

When in state $\boldsymbol{s}$, a player u can select an action vector $\boldsymbol{a} \in \mathcal{A}$ of size m that gathers the "production orders". The j^{th} component of this vector corresponds to the production variable s_{p_j}. When an action $\boldsymbol{a}$ is taken, the production variables of $\boldsymbol{s}$ are immediately modified according to:

$$\forall j \in \{1, ..., m\} : s_{p_j} \leftarrow s_{p_j} + a_j$$

We define a production strategy for player u as a mapping $P : \mathcal{S} \rightarrow \mathcal{A}$ which selects an action vector $\boldsymbol{a}$ for any given state vector $\boldsymbol{s}$:

$$\boldsymbol{a} = P(\boldsymbol{s})$$

V. Learning Architecture

We assume that a set of recorded games constituted of state vectors $\boldsymbol{s}^u \in \mathcal{S}^u$ of player u is provided. Our objective is to learn the production strategy P^u used by player u. To achieve this, we use supervised learning to learn to predict each production variable s_{p_j} based on the remaining state $\boldsymbol{s}_{-p_j}$ defined below. We then use the predicted s_{p_j} values to deduce a production order $\boldsymbol{a}$. Since there are m production variables, we solve m supervised learning problems. Formally, our approach works as follows.

For any state vector $\boldsymbol{s}$, we define the remaining state for each production variable s_{p_j} as $\boldsymbol{s}_{-p_j}$:

$$\forall j \in \{1, ..., m\} : \boldsymbol{s}_{-p_j} = (s_1, s_2, ..., s_{p_j - 1}, s_{p_j + 1}, ..., s_n)$$

For each production variable, we define a learning set $\{(\boldsymbol{s}^u_{-p_j}, s^u_{p_j})\}_{\boldsymbol{s}^u \in \mathcal{S}^u}$ from which we learn a function $\hat{P}^u_j$ which maps any remaining state $\boldsymbol{s}_{-p_j}$ to a unique $\hat{P}^u_j(\boldsymbol{s}_{-p_j})$. Knowing each $\hat{P}^u_j$, we can deduce a mapping $\hat{P}^u$ and estimate a production order $\boldsymbol{a}$ for any given state vector $\boldsymbol{s}$:

$$\begin{aligned}\boldsymbol{a} = \hat{P}^u(\boldsymbol{s}) = (&\hat{P}^u_1(\boldsymbol{s}_{-p_1}) - s_{p_1}, \\ &\hat{P}^u_2(\boldsymbol{s}_{-p_2}) - s_{p_2}, ..., \hat{P}^u_m(\boldsymbol{s}_{-p_m}) - s_{p_m})\end{aligned}$$

Using this approach, we learn the production strategy used by player u by learning m $\hat{P}^u_j$ functions to estimate production variables given the remaining state variables. Each $\hat{P}^u_j$ is learned separately using supervised learning. In other words, we learn m models. For each model, the input for the learning algorithm is the state vector $\boldsymbol{s}$ stripped from the component the model must predict, which becomes the output. This process is illustrated in Figure 2.

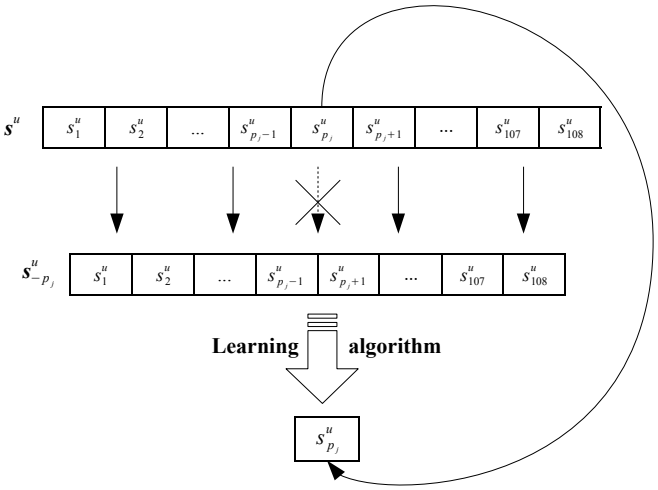


Fig. 2. Learning the p_jth model

It is worth stressing that the action vector $\boldsymbol{a}$ computed by the mapping $\hat{P}^u$ learned may not correspond to, due to the constraints imposed by the game, an action that can be taken. For example, $\boldsymbol{a}$ may send among others an order for a new type of unit while the technology it requires is not yet available. In our implementation, every component of $\boldsymbol{a}$ which is inconsistent with the state of the game is simply set to zero before the action vector is applied.

VI. Experimental Results

The proposed method was tested in StarCraft II by teaching a Terran agent facing a Protoss opponent production strategies.

A total of $n = 108$ variables were selected to describe a state vector. These state variables are:

- $s_1 \in \mathbb{N}$ is the time elapsed since the beginning of the game in seconds
- $s_2 \in \mathbb{N}$ is the total number of units owned by the agent
- $s_3 \in \mathbb{N}$ is the number of SCVs (Space Construction Vehicles)
- $s_4 \in \mathbb{N}$ is the average mineral harvest rate in minerals per minute
- $s_5 \in \mathbb{N}$ is the average gas harvest rate in gas per minute

- $s_u \in \mathbb{N}, u \in \{6, ..., 17\}$ is the cumulative number of units produced of each type
- $s_b \in \mathbb{N}, b \in \{18, ..., 36\}$ is the number of buildings of each type
- $s_t \in \{0, 1\}, t \in \{37, ..., 63\}$ indicates whether each technology has been researched
- $s_e \in \{0, 1\}, e \in \{64, ..., 108\}$ indicates whether an enemy unit type, building type or technology has been encountered

Among these, there are $m = 58$ variables which correspond to direct production orders: 12 s_u unit variables, 19 s_b building variables and 27 s_t technology variables. Therefore, an action vector is composed of 58 variables. These action variables are:

- $a_u \in \mathbb{N}, u \in \{1, ..., 12\}$ corresponds to the number of additional units of each type the agent should produce
- $a_b \in \mathbb{N}, b \in \{13, ..., 31\}$ corresponds to the number of additional buildings of each type the agent should build
- $a_t \in \{0, 1\}, t \in \{32, ..., 58\}$ corresponds to the technologies the agent should research

The Terran agent learned production strategies from a set of 372 game logs generated by letting a Very Hard Terran computer player (u) play against a Hard Protoss computer player on the Metalopolis map. State vectors were dumped every 5 seconds in game time. Each $\hat{P}_j^u$ was learned using a feedforward neural network with a 15-neuron hidden layer and the Levenberg-Marquardt backpropagation algorithm [15] to update weights. Inputs and outputs were mapped to the $[-1, 1]$ range. A tan-sigmoid activation function was used for hidden layers.

Because it is not possible to alter production decisions in the Very Hard Terran player without giving up the remaining non production decisions, these 58 neural networks were combined with a simple scripted combat manager which handles when the agent must attack or defend. On the other hand, the low level unit AI is preserved. During a game, the agent periodically predicts production orders. For any given building type, unit type or technology, if the predicted target value $\hat{P}_j^u(\boldsymbol{s}_{-p_j})$ is greater than the current number s_{p_j}, a production order a_j is passed to reach the target value. This behavior is illustrated in Figure 3.

The final agent was tested in a total of 50 games using the same settings used to generate the training set. The results are summarized in Table 1. With a less sophisticated combat handler, the imitative learning trained agent (IML agent) managed to beat the Hard Protoss computer player 9 times out of 10 on average while the Hard Terran computer player lost every game. This performance is not far below that of the Very Hard Terran computer player the agent learned from, which achieved an average win rate of 96.5%. In addition to counting victories, we have attempted to verify that the agent indeed replicates to some extent the same production strategies as those from the training set. Roughly, two different strategies were used by the Very Hard Terran computer player. The first one (A) primarily focuses on infantry while the second one (B) aims at faster technological development. Formally, a game is given the label Strategy A if no factories or starports are built during the first 5 minutes of the game. Otherwise it is labeled Strategy B. Figure 4 shows, for the training set, the average number of barracks, factories and starports built over time for each strategy. Two corresponding strategies were also observed for the learning agent over the 50 test games, as shown in Figure 5. For each strategy, the frequency of appearance is shown in Figure 6.

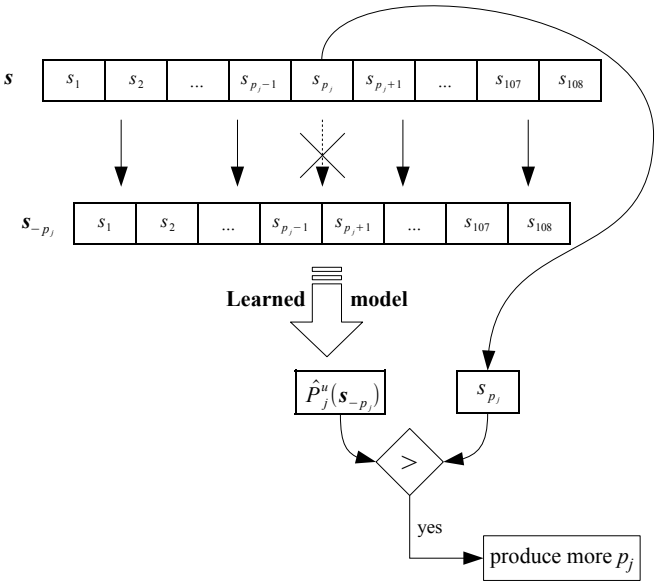


Fig. 3. Agent production behavior

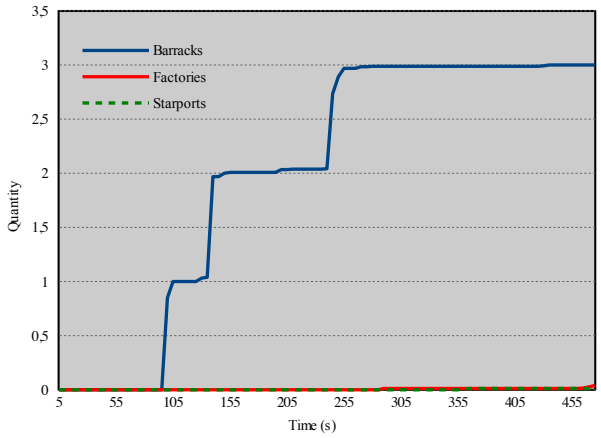
TABLE I
TERRAN PERFORMANCE AGAINST HARD PROTOSS

	Terran win rate	**Total games**
Very Hard Terran	96.5%	372
Hard Terran	0%	50
IML agent	**90%**	**50**

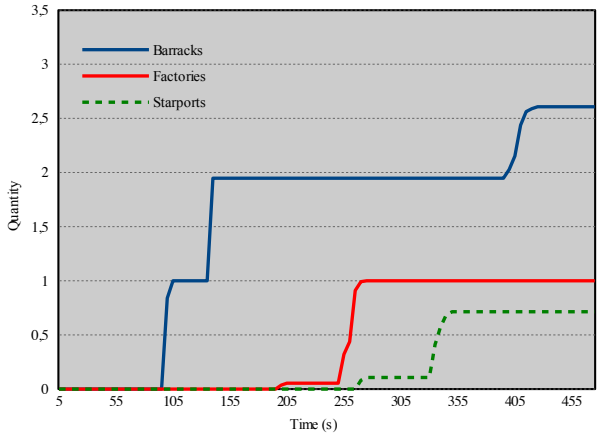
The frequency at which each strategy is used was not faithfully reproduced on the test set. This can be partly explained by the more limited combat handler, which may fail to acquire the same information on the enemy than was available in the training set. Moreover, Strategy B seems to be less accurately replicated than Strategy A. This may be caused by the lower frequency of appearance in the training set. Nevertheless, the results obtained indicate that the agent learned both production strategies from the Very Hard Terran computer player. Subsequently, we may rightly attribute the agent's high performance to the fact that it managed to imitate the efficient production strategies used by the Very Hard Terran computer player.

VII. Conclusion and Future Work

In this paper, we have presented a method for integrating imitative learning in real-time strategy agents. The proposed solution allowed the creation of an agent for StarCraft II capable of learning production strategies from recorded game

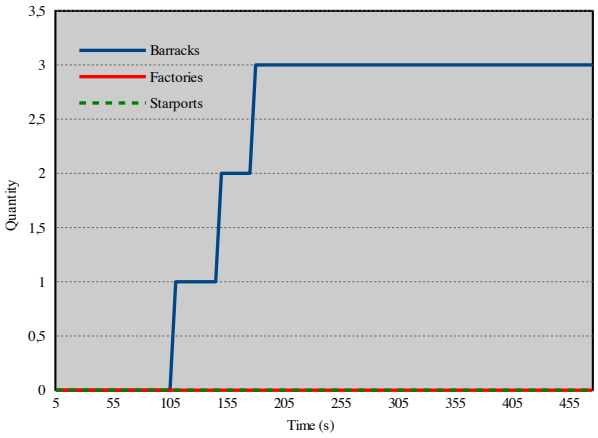


(a) Strategy A

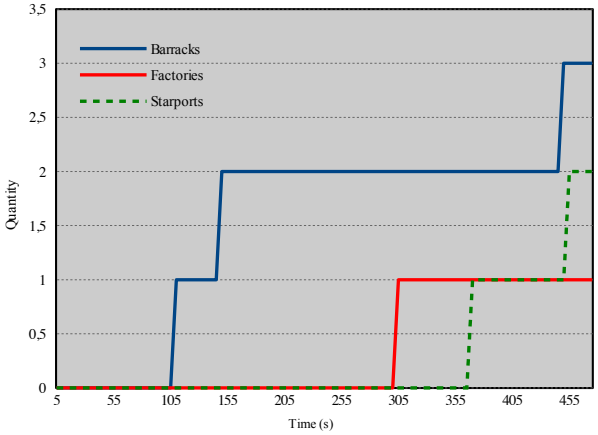


(b) Strategy B

Fig. 4. Training set strategies



(a) Strategy A



(b) Strategy B

Fig. 5. Test set strategies

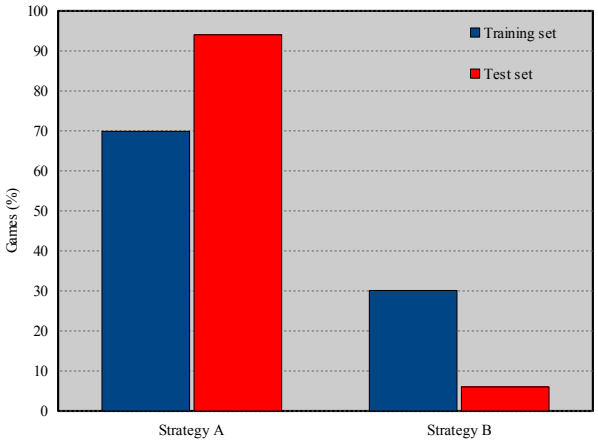


Fig. 6. Strategy frequencies

data and applying them in full one-on-one games. However, since the training data was artificially generated, the agent is restricted to a specific matchup type. A larger and more diverse dataset would be required to significantly impact the performance of agents against human players. We therefore plan on extending this work to larger datasets.

In order to efficiently learn from richer sets, potentially collected from various sources, we suspect clustering will be required to organize records and maintain manageable datasets. Furthermore, the manually generated training data only contained desirable production strategies. When training data is automatically collected from various sources, selection techniques will be required to filter out undesirable production strategies. We believe that with a large enough set, the learned production strategy models should be robust enough to be used against human players.

Besides production-related improvements, there are other areas worth investing in to increase agent performance such as information management or combat management. Enhanced information management can allow an agent to better estimate the state of its opponents and for example predict the location of unit groups that could be killed before they can retreat or be joined by backup forces. As for combat management, it may lead to much more efficient unit handling in battle and for example maximize unit life spans.

ACKNOWLEDGMENTS

Raphael Fonteneau is a postdoctoral researcher of the FRS-FNRS from which he acknowledges the financial support. This paper presents research results of the Belgian Network DYSCO (Dynamical Systems, Control, and Optimization),

funded by the Interuniversity Attraction Poles Programme, initiated by the Belgian State, Science Policy Office. The scientific responsibility rests with its author(s).

Legal Notice

References

[1] I. Umarov, M. Mozgovoy, and P. C. Rogers, "Believable and effective AI agents in virtual worlds: Current state and future perspectives," *International Journal of Gaming and Computer-Mediated Simulations*, vol. 4, no. 2, pp. 37–59, 2012.

[2] J. Togelius, G. N. Yannakakis, S. Karakovskiy, and N. Shaker, "Assessing believability," *Believablility in Computer Games*, 2011.

[3] J. E. Laird and van Michale Lent, "Human-level AI's killer application: Interactive computer games," in *Proceedings of the 17th National Conference on Artificial Intelligence*, 2000.

[4] C. Bauckhage, C. Thurau, and G. Sagerer, "Learning human-like opponent behavior for interactive computer games," *Pattern Recognition*, pp. 148–155, 2003.

[5] B. Gorman and M. Humphrys, "Imitative learning of combat behaviours in first-person computer games," in *Proceedings of the 10th International Conference on Computer Games: AI, Mobile, Educational and Serious Games*, 2007.

[6] C. Thurau, G. Sagerer, and C. Bauckhage, "Imitation learning at all levels of game AI," in *Proceedings of the International Conference on Computer Games, Artificial Intelligence, Design and Education*, 2004.

[7] F. Frandsen, M. Hansen, H. Sørensen, P. Sørensen, J. G. Nielsen, and J. S. Knudsen, "Predicting player strategies in real time strategy games," Master's thesis, 2010.

[8] S. Ontañón, K. Mishra, N. Sugandh, and A. Ram, "Case-based planning and execution for real-time strategy games," in *Proceedings of the 7th International Conference on Case-Based Reasoning (ICCBR-07)*, 2007, pp. 164–178.

[9] D. W. Aha, M. Molineaux, and M. J. V. Ponsen, "Learning to win: case-based plan selection in a real-time strategy game," in *Proceedings of the 6th International Conference on Case-Based Reasoning (ICCBR-05)*, 2005, pp. 5–20.

[10] B. Weber and M. Mateas, "Case-based reasoning for build order in real-time strategy games," in *Proceedings of the 5th Artificial Intelligence for Interactive Digital Entertainment Conference (AIIDE-09)*, 2009.

[11] B. G. Weber and M. Mateas, "A data mining approach to strategy prediction," in *Proceedings of the 5th IEEE Symposium on Computational Intelligence and Games (CIG-09)*. IEEE Press, 2009, pp. 140–147.

[12] M. Ponsen, H. Muñoz-Avila, P. Spronck, and D. Aha, "Automatically generating game tactics via evolutionary learning," *AI Magazine*, vol. 27, no. 3, pp. 75–84, 2006.

[13] J. McCoy and M. Mateas, "An integrated agent for playing real-time strategy games," in *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI-08)*, 2008, pp. 1313–1318.

[14] F. Safadi, R. Fonteneau, and D. Ernst, "Artificial intelligence design for real-time strategy games," in *Proceedings of the 2nd International Workshop on Decision Making with Multiple Imperfect Decision Makers*, 2011.

[15] M. I. A. Lourakis, "A brief description of the Levenberg-Marquardt algorithm implemened by levmar," 2005.

[16] D. C. Cheng and R. Thawonmas, "Case-based plan recognition for real-time strategy games," in *Proceedings of the 5th International Conference on Computer Games: Artificial Intelligence, Design and Education (CGAIDE-04)*, 2004, pp. 36–40.

[17] M. Chung, M. Buro, and J. Schaeffer, "Monte-Carlo planning in real-time strategy games," in *Proceedings of the 1st IEEE Symposium on Computational Intelligence and Games (CIG-05)*, 2005.

[18] A. Kovarsky and M. Buro, "A first look at build-order optimization in real-time strategy games," in *Proceedings of the 2006 GameOn Conference*, 2006, pp. 18–22.

[19] B. G. Weber and M. Mateas, "Conceptual neighborhoods for retrieval in case-based reasoning," in *Proceedings of the 8th International Conference on Case-Based Reasoning (ICCBR-09)*, 2009, pp. 343–357.

[20] S. Schaal, "Is imitation learning the route to humanoid robots?" *Trends in Cognitive Sciences*, vol. 3, no. 6, pp. 233–242, 1999.

[21] R. M. Gray, "Vector quantization," *IEEE ASSP Magazine*, vol. 1, p. 4, 1984.

[22] F. V. Jensen and T. D. Nielsen, *Bayesian Networks and Decision Graphs*. Springer Science+Business Media LLC, 2007.