Diogo Rafael Cordeiro Alves

MODELING OF SYNTHETIC PLAYERS AS AN INSTRUMENT FOR TESTING GENERATIVE CONTENT

1 2 9 0 UNIVERSIDADE Đ COIMBRA

1 2 9 0

UNIVERSIDADE Đ
COIMBRA

Diogo Rafael Cordeiro Alves

MODELING OF SYNTHETIC PLAYERS AS AN INSTRUMENT FOR TESTING GENERATIVE CONTENT

Dissertation in the context of the Master in Informatics Engineering, Specialization in Intelligent Systems, advised by Professor Licínio Roque, and presented to Faculty of Sciences and Technology / Department of Informatics Engineering.

June 2021

Faculty of Sciences and Technology

Department of Informatics Engineering

Modeling of Synthetic Players As An Instrument For Testing Generative Content

Diogo Rafael Cordeiro Alves

Dissertation in the context of the Master in Informatics Engineering, Specialization in Intelligent Systems advised by Prof. Licínio Roque and presented to the Faculty of Sciences and Technology / Department of Informatics Engineering.

June 2021

1 2 9 0

UNIVERSIDADE Ð COIMBRA

This page is intentionally left blank.

Abstract

There is a need to discover new, reliable techniques to test procedurally generated game scenarios. One method that has potential is automated testing, which consists in using synthetic players to play and test the newly generated scenarios. Therefore, to test this assumption, this work models two types of synthetic players that are put in a *Bomberman-like* scenario. In this case study, the players must place bombs in strategic places in order to eliminate the opponents. One synthetic player is developed via a planning-based approach, in which the agent searches for a sequence of actions that leads from the current state of the game to the desired game state. The second approach is machine learning, more precisely, combining Imitation Learning with Reinforcement Learning in order for the synthetic player to first learn by observing human demonstrations, and then improve its performance by maximizing its policy via the environment rewards. Results show that the planning synthetic player manages to play and win the game consistently against opponents developed by the machine learning approach, and can generalize well to new unseen scenarios. It also obtained positive scores in a survey regarding its believability. All of these attributes makes the planning synthetic player a viable tool to test procedurally generated game scenarios. However, the planning agent struggles when facing a human player, providing an easy-to-moderate challenge for the human. The machine learning approach produced modest results, not being able to win the game consistently. Its modest performance harms its believability as well. Nonetheless, a particular trained model could be perceived as if grasping how to play the game and managed to survive long-enough periods to explore the game space.

Keywords

Artificial Intelligence in Games, Synthetic Player, Imitation Learning, Reinforcement Learning, Planning

This page is intentionally left blank.

Resumo

Há uma necessidade de se encontrar novos métodos confiáveis para testar cenários de jogo gerados procedimentalmente. Um método que mostra potencial é teste automatizado, que consiste em usar jogador sintéticos para testar novos cenários de jogo. Deste modo, para testar esta suposição, este trabalho modela dois tipos de jogadores sintéticos que são colocados num cenário semelhante ao *Bomberman*. Neste caso de estudo, os jogadores têm de colocar bombas em lugares estratégicos para eliminar os adversários. Um jogador sintético é desenvolvido através de uma abordagem baseada em planeamento, em que o agente procura por uma sequência de ações que leva desde o estado atual do jogo até ao estado desejado. A segunda abordagem é aprendizagem computacional, mais precisamente combinar aprendizagem por imitação com aprendizagem por reforço, para que primeiro, o jogador sintético aprenda observando demonstrações humanas, e depois melhore a sua performance através das recompensas do ambiente. Resultados mostram que o jogador sintético de planeamento consegue jogar e ganhar o jogo consistentemente contra oponentes desenvolvidos pela abordagem de aprendizagem computacional, e consegue generalizar bem para novos cenários. Também obteve resultados positivos num inquérito acerca da sua credibilidade. Estes atributos fazem do jogador sintético de planeamento uma ferramente viável para testar cenários de jogo gerados procedimentalmente. Contudo, o agente de planeamento tem dificuldades quando joga contra um jogador humano, fornecendo apenas um desafio de dificuldade fácil a moderada para o humano. A abordagem de aprendizagem computacional produziu resultados modestos, não sendo capaz de vencer o jogo consistentemente. O seu desempenho pouco satisfatório também magoa a sua credibilidade. Não obstante esse facto, um modelo treinado aparenta ter aprendido as regras básicas do jogo e conseguiu sobreviver períodos de tempo suficientes para explorar o espaço de jogo.

Palavras-Chave

Inteligência Artificial em Jogos, Jogador Sintético, Aprendizagem por Imitação, Aprendizagem por Reforço, Planeamento

This page is intentionally left blank.

Agradecimentos

Primeiro de tudo, quero agradecer ao meu orientador, Professor Doutor Licínio Roque, pela sua supervisão e pela oportunidade que me conferiu de trabalhar sobre a área de investigação que mais me cativa, IA em Jogos. De seguida, quero agradecer aos júris da minha dissertação, Professor Doutor Ernesto Costa e Professor Doutor Paulo Simões, pelas recomendações fornecidas que ajudaram a guiar o meu trabalho.

Um obrigado especial ao Bernardo, por providenciar a plataforma e os cenários de jogo essenciais ao desenvolvimento do meu trabalho, e também à Kasia, por ter dedicado muito do seu tempo a ajudar-me a melhorar as minhas capacidades de inglês.

Finalmente, como não podia deixar de ser, quero agradecer aos meus pais e aos meus amigos por todo o apoio que me deram ao longo desta jornada.

Diogo Rafael Cordeiro Alves

Espinhal, Junho de 2021

Contents

Acronyms

AI Artificial Intelligence. 3, 18–21, 24, 31

BC Behavioral Cloning. 36, 37, 45, 53, 55–60, 62–65, 67, 68, 78

CGI Computer-Generated Imagery. 25

GAIL Generative Adversarial Imitation Learning. 29, 45, 51, 53, 55–60, 62, 63, 65, 67, 78

IL Imitation Learning. xv, 4, 6, 34, 45, 46, 51, 53, 56, 58, 66, 67, 78

LSTM Long-Short Term Memory. 37

MCTS Monte Carlo Tree Search. 17, 23

NPC Non-Playable Character. 1, 13, 19–21, 25, 30, 31

PCG Procedural Content Generation. 1, 2, 7, 71, 83

PPO Proximal Policy Optimization. 36, 37, 45, 46, 56

RL Reinforcement Learning. viii, 4, 6, 7, 18, 34, 45–47, 49, 51, 56–58, 60, 66–68, 78

SAC Soft Actor Critic. 45, 46

UCB Upper Confidence Bound. 23

List of Figures

This page is intentionally left blank.

List of Tables

This page is intentionally left blank.

Chapter 1

Introduction

1.1 Motivation

Some academics and game developers claim that Non-Playable Character (NPC) behaviour is already “solved” [Nareyek, 2007]. By this, they mean that the current performance of NPCs in games has already reached highly satisfactory results, and there is not a lot more to be done to improve the area of NPC modeling even further (at least, without allocating a significant amount of resources to it). This claim is supported by statistical proof. Analyzing the trending topics of main conferences like IEEE CIG (Computational Intelligence and Games), AIDE (Artificial Intelligence and Interactive Digital Entertainment), and FDG (Foundation of Digital Games), one can clearly see that NPC behavior and game-playing agents fell in the pecking order, with areas such as Procedural Content Generation (PCG) and player modeling becoming more popular among researchers [Yannakakis and Togelius, 2015]. Therefore, there is indeed a shift of mentality towards other areas of using AI in games, besides NPC-related matters [Yannakakis, 2012]. Despite these areas’ rise in popularity, research about the relationship between them is sparser. For instance, exploring the use of agents in procedural content generation is a field vastly underexplored, although the potential of said relationship is already known [Yannakakis and Togelius, 2015].

In fact, one of the open challenges of procedural content generation is the lack of reliable techniques to validate the generated content. This is particularly evident in the game industry, where modern videogames are relying more and more on procedural generation techniques to create a large amount of content, like vast game worlds. Game playtesters are not able to test everything that is procedurally created in time, before the game is publicly released. This situation may lead to unwanted results, e.g. launching a defective product in the market.

Therefore, there is a need to discover new reliable techniques that improve the game testing process. One method that shows promise is automated playtesting. It consists of using agents (synthetic players) that can faithfully replace a human playtester. This still underexplored approach is the subject matter of this work.

1.2 Goals and Contributions

This dissertation aims to show the potential value of using agents as instruments in playing and testing procedural generated content. To achieve this, two types of synthetic players

are developed and evaluated on game scenarios generated via PCG.
In order to successfully develop a synthetic player, three main requirements must be fulfilled [Shih and Teng, 2020]:

- **Ability to play and win the game**. First and foremost, synthetic players need to be able to play and win the game, since their main goal is to play and test game scenarios.
- **Ability to play new untested game scenarios**. In order to be reliable automated testing tools, synthetic players need to be able to play previously unseen game scenarios. This translates into the need for a good **generalization** capability - the process of extrapolating previously obtained knowledge to a new scenario in order to adapt to the unknown.
- **Playing in a believable way**, i.e. in a human-like manner. An agent used in automated testing needs to act in a believable manner, otherwise the feedback from the agent testing a certain game scenario may be untrustworthy. If this requirement is not met, an agent can exploit the game to win instead of playing it as intended.

With these requirements in mind, two AI techniques were chosen to model the synthetic players - planning and machine learning. The choice of using these two particular methods is justified in section 3.1, weighing their pros and cons compared to other AI techniques. Therefore, the contributions of this dissertation are:

- An implementation of a planning agent, that searches its action space to find a sequence of actions that achieves the current goal;
- An implementation of a reinforcement learning agent, that first learns from demonstrations of gameplays of a human player and then keeps improving its performance due to extrinsic rewards.
- An analysis of the performance of both implementations and their potential success in fulfilling the three requirements for modeling synthetic players listed above;

The synthetic players will act upon procedurally generated game scenarios of a turn-based adaptation of the Bomberman video game, which will be explained in detail in section 3.2. The development of the game system and the procedural generation of game scenarios are out of the scope of this dissertation. The game scenarios and the platform itself where synthetic players will be tested are an output of the works of another dissertation: *Modeling and Generation of Playable Scenarios Based on Societies of Agents* [Correia, 2021].

1.3 Methodologies

In the making of this project, a Design Science Research-based methodology was applied. This process is composed of five stages. First, the Awareness of a Problem, where one discovers and defines a particular problem and tries to understand it. In the context of this dissertation, the problem was already identified: the potential of using agents as automated testing and game-playing tools, and how underdeveloped this area of research is. In addition, the problem was also formally defined, within the list of the goals of this dissertation, in 1.2.

With the problem initially understood, the next step is to generate a solution proposal in the Suggestion stage. After the theoretical conception of the proposal (agent representation and choice of techniques), one proceeds to the Development phase that generates an artifact (synthetic players). In this context, the artifact is a proof of concept of the proposed solution. Next, in order to verify if the developed prototype's performance meets the expectations, one proceeds to the Evaluation phase (running a simulation to collect empirical data).

Design Science Research is an iterative process because knowledge obtained from the Development and Evaluation phases might reveal that the current suggestion is inappropriate or even obsolete. Therefore, there is a necessity of going back to the suggestion phase, to propose a different approach in the light of the newly obtained knowledge.

Finally, when the results of the evaluation phase are satisfactory, one may move on to the last stage - the Conclusion, where the final results are presented and a reflection about all the elaborated work is made. In most cases this is attained only after several iterations of the whole process. After generating the final artifacts (models of synthetic players), this dissertation is completed. Figure 1.1 allows the reader to better visualize the flow of the Design Science Research process.

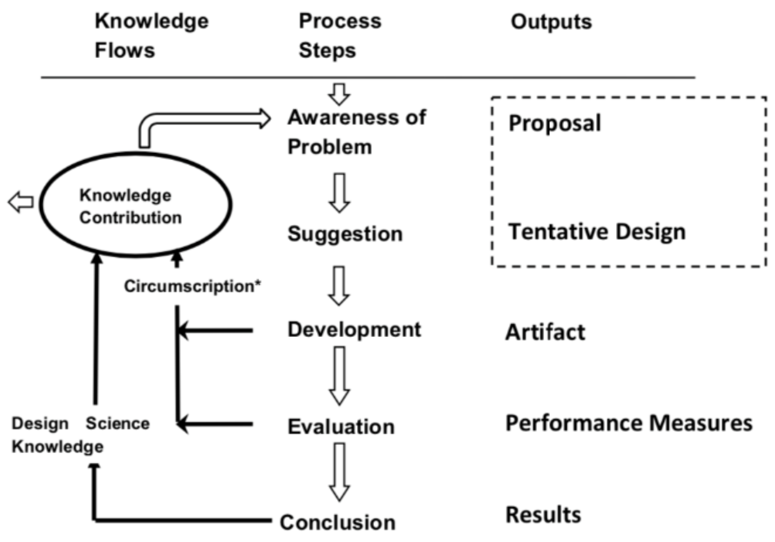

Figure 1.1: Design Science Research process [Vaishnavi et al., 2004]. *Circumscription is the formal definition for the assumption that things work as expected unless stated otherwise.

During the Development Phase, since it is the largest of all, some practices of the Scrum software engineering methodology were applied, namely a temporal division of the development tasks in sprints and the existence of both a product and a sprint backlog. A product backlog is a list of all the tasks needed to be done in order for work to be fully completed. A sprint backlog is a list of tasks chosen from the product's backlog to be implemented during the specified sprint.

Sprints were defined as either a half of or a full calendar month. An initial design proposal was made and exposed in this dissertation's intermediate defense at the end of January. Taking into account the recommendations obtained in the defense, the entire February was dedicated to changing and tweaking the design proposal. Therefore, the development phase started effectively in the beginning of March.

Thus, the first sprint occurred during the entire March, and its backlog consisted of two high-level tasks (decomposed in several minor tasks): development of a prototype (to be used as a temporary platform) for the selected case study and implementation of the first Artificial Intelligence (AI) approach to model synthetic players, i.e., the planning agent.

While the temporary platform was completed, the planning agent implementation was not finished in the first sprint and therefore went back to the product backlog again.

The second sprint was composed by the first half of April, until the proper platform for the case study was finished by [Correia, 2021], and its backlog had two high-level tasks to be completed: the development of the planning agent that was not finished in the first sprint, and an initial evaluation of its performance. Results suggested that the implementation needed some additions and changes, and therefore, the task regarding the development of the agent was once again put back in the product backlog.

The third sprint encompassed the 2nd half of April, and the high-level tasks selected for its backlog were transitioning the planning agent to the real platform and setting up the machine learning agent to be ready for training.

Fourth sprint comprised the first half of May, and was dedicated to machine learning agent's training (by IL and classical Reinforcement Learning (RL)), and an initial evaluation of its performance. Results showed that more training was required in order for the agent to achieve relevant results.

In the fifth sprint, which occurred during the last two weeks of May, the planning agent-related task was completed, and more machine learning training was planned. An evaluation phase, comparing the performance of both agents, was done. Results showed that not only was the machine learning agent not ready yet, but also that a few minor tweaks to the planning agent were to be done. Since the initial sprint changes of the planning synthetic player took less time than expected, it was also possible to implement the new needed tweaks that had resulted from the previous Evaluation phase in the same sprint.

The last sprint was composed of the first two weeks of June, when the machine learning agent trained once again, and then evaluation was done in the middle of the sprint. Results were still not relevant enough, so a new training approach was devised and the agent started to train in a self-play environment, with a curiosity module incorporated as well.

A final Evaluation phase of the performance of all the modelled synthetic players and conclusion of the work happened in the last two weeks of June.

1.4 Workplan

The effective work for this dissertation during the first semester is depicted in Figure 1.2. It was mostly composed of gathering and reading research material, and then devising an initial design proposal to address the stated problem.

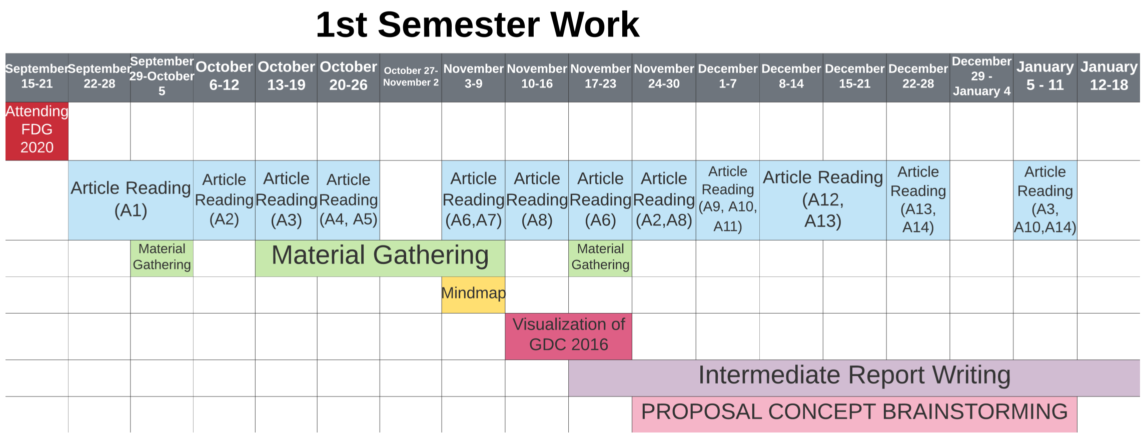

Figure 1.2: Gantt Diagram of the work done in the first semester.

Weeks in Figure 1.2 are considered to start on Tuesdays, for one main reason: the weekly meeting with the supervisor usually happens on Tuesdays, and sets the pace for the rest of the week. Furthermore, the work officially started with attending the Foundations of Digital Games 2020 Conference (FDG 2020), which also happened to start on a Tuesday. Finally, the fact that the report submission deadline (January 18) was on a Monday was a fortunate coincidence that further supported the decision of considering the beginning of the working week on Tuesdays and the end of it on Mondays. The *Article Reading* tasks are as follows:

- **A1**- James Ryan's PHD Dissertation;
- **A2**- *Player Modeling* - Chap. 5 of *Artificial Intelligence and Games* Book [Yannakakis and Togelius, 2018];
- **A3**- AI in Games: Methods and Subareas of Research;
- **A4**- A. Loyall's PHD Dissertation;
- **A5**- NLP Articles;
- **A6**- Believable Agents Articles;
- **A7**- *AI - A Modern Approach Book* [Russell and Norvig, 2010];
- **A8**- *AI Playing The Game* - Chap.3 of *Artificial Intelligence and Games* Book [Yannakakis and Togelius, 2018];
- **A9**- *Growing Artificial Societies* Book [Epstein, 1994];
- **A10**- Disease Modeling Articles;
- **A11**- Player Modeling Articles;
- **A12**- Simulation-Based Testing;
- **A13**- *AI And Artificial Life in Video Games* Book [Lecky-Thompson, 2008];
- **A14**- Agent-Based Modeling;

Adopting the Design Science Research methodology described previously, in the end of the project, the effective work done during second semester is depicted in Fig 1.3.

The tasks depicted in Figure 1.3 are the following:

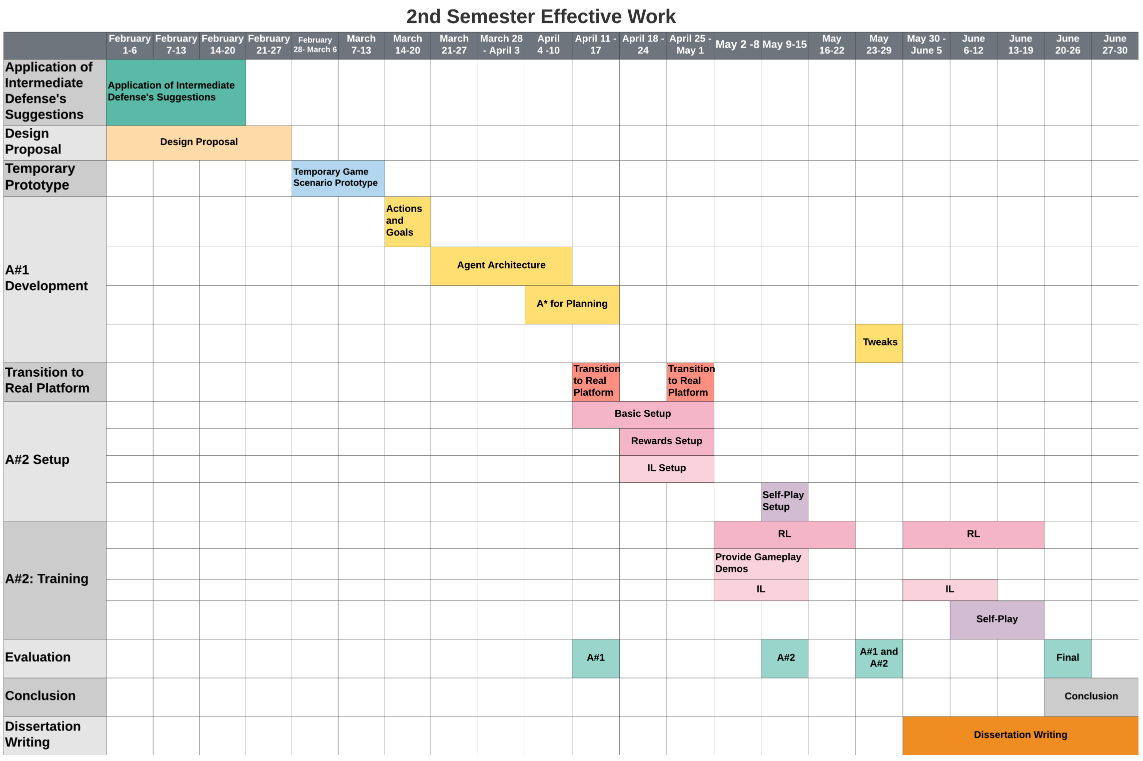

Figure 1.3: Gantt Diagram of work done during the second semester. (A#1 denotes the planning synthetic player approach; A#2 represents the machine learning approach)

- **Application of Intermediate Defense's Suggestions**: Incorporating the recommendations provided at the intermediate defense to the document.
- **Design Proposal**: Conception of the design proposal to be implemented and tested in order to solve the identified problem.
- **Temporary Prototype**: Development of a temporary game scenario prototype in order for the work to progress, while waiting for the conclusion of the actual platform.
- **A#1 Development**: Development of the planning approach, that was decomposed in several subtasks (definition of actions and goals, building the agent model, implementing the A* Algorithm for the planning process, and final tweaks to the agent).
- **Transition to Real Platform**: Any task that is performed in order to adjust the implementation work so far to the final platform.
- **A#2 Setup**: Setup of the machine learning approach. This includes setting up the basic RL loop, implementing the functions to calculate rewards and setup of the heuristic mode of the agent to be able to provide gameplay demonstrations in order for it to perform IL.
- **A#2 Training**: All the training tasks referring to the training process of the machine learning agent. This includes providing gameplay demonstrations, IL and RL training.
- **Evaluation**: Refers to the experiments done in order to evaluate the performance of the developed synthetic players.
- **Conclusion**: Summarizing the results of the final evaluation phase, discussing them, and finishing the entire project.
- **Dissertation Writing**: Writing the document.

1.5 Tools

The Unity game engine [1] was used for the development of the project. Unity is a cross-platform game engine that allows the creation of both 2D and 3D worlds, and is popular among game developers. It supports scripting in C#. Since one approach to model synthetic players is machine learning, a special framework is required. Unity provides the ML-Agents toolkit [2] [3], an open-source framework that allows to train intelligent agents with state-of-art reinforcement learning algorithms in an Unity environment. Finally, Github [4] is also used for version control and to easily merge contributions of both dissertations (this one and the one of [Correia, 2021]).

1.6 Risks Assessment

This project was subjected to some possible risks:

- **R1**: The game scenarios, as the platform itself, are being developed concurrently; This dependency can impact the timeline of the project.

Mitigation strategy: Close collaboration and information sharing to enable timeline adjustments. The development process was planned in cooperation and some Scrum practices are adopted (iterations, sprints, planning and tracking meetings) to cope with unforeseen conditions.

- **R2**: The game scenarios, as the platform itself, are being developed concurrently; The inadequacy of the synthetic players being tested for the PCG game scenarios under development could endanger the results.

Mitigation strategy: Close collaboration and information sharing to specify useful PCG testing scenarios.

- **R3**: The machine learning framework that is used is a black-box type, not allowing to twinkle with the neural network architecture; It makes it difficult to recognise problems with the agent's training and make adaptations.

Mitigation strategy: Dealing with the black-box nature of the framework is hard and there is not much that can be done about it. This risk became an actual problem due to the lack of a mitigation measure addressed to it. A combination of planning (white box) and RL (black box) strategies was tested to minimize dependency and enable more control over the study conditions. A series of testing cases enabled analysing ways of making progress empirically.

[1]Unity Official Page: https://unity.com/
[2]ML-Agents Framework Official Page: https://unity.com/products/machine-learning-agents
[3]ML-Agents Framework Github: https://github.com/Unity-Technologies/ml-agents
[4]Github Offical Page: https://github.com/

1.7 Structure Of This Document

The outline of the rest of this document is as follows.
Chapter 2 describes the state-of-the-art of using AI to model agents to play games and to act in a believable way.
Chapter 3 unveils the implementation details of the developed synthetic players, while **Chapter 4** lists the experiments done in order to evaluate their performances.
This work ends with **Chapter 5**, which is a conclusion of the obtained results and states its contributions for future research.

Chapter 2

State-of-the-Art

In this section, an in-depth overview of how AI is used to play games is given, answering the questions “why games for AI?”, “What type of games can AI play?” and “How can AI play games?”. Then, the section summarizes the state-of-art of simulation-based testing. As already said in section 1.2, one of the requirements to model synthetic players is the believability of their behaviours. Henceforth, this section also approaches the field of believable agents, and finishes with how player modeling can also be used to model said agents.

2.1 AI Playing the Game

This chapter discusses the use of Artificial Intelligence in games, more specifically, to play a game. It starts with the explanation of why the relationship between games and AI is beneficial, then lists the type of games that AI can play and how. The section then summarizes which roles AI can assume while playing the game.

2.1.1 Why Games For AI

The benefits of AI in games are clear: AI techniques can be used to play the game, which in turn helps to improve the overall experience of the human player [Yannakakis and Togelius, 2018]. The subarea of procedural content generation can also be used to diminish the game designer’s authorial burden. However, what about the inverse? What is the motivation for bringing games to the field of Artificial Intelligence? In fact, there are several strong reasons for such symbiosis.

Games are hard and interesting problems, even the ones that have simple rules [Yannakakis and Togelius, 2018]. This may sound paradoxical, but indeed, most games can be represented mathematically in just a few lines while at the same time having a large search space, which is always a hard problem to solve. One example of such a game is the ancient board game *Go*, that despite its simple rules, it has a search space of 10^{270} states [Russell and Norvig, 2010].

Understandability because it can easily be seen what the agent is doing and how well it is performing[Schaul et al., 2011].

Public awareness because most people know what games are about, so a research breakthrough is easier to explain by extrapolating it to a game than explaining it in an abstract manner like theorem proving[Schaul et al., 2011].

Speed is also a reason due to the possibility of speeding up the simulation of games, and therefore being able to play a game several hundred times per second[Schaul et al., 2011].

Industrial Application The game industry can be an economically very relevant application domain for AI, since it is the largest entertainment industry[Schaul et al., 2011]. This factor also contributes to the sheer amount of content that exists regarding games. Since there is more content, that leads to a big diversity as well. In fact, there are a plethora of game genres, from classic arcade games to the modern ones that present complex open-world *3D* environments. Therefore, this wide diversity is also an important factor to support the use of games as AI testbeds since it provides challenges to all AI areas. Natural language processing, machine learning, search and planning, knowledge representation, and reasoning, among many other areas, have reached breakthroughs using games to measure their performance.

Notable Examples

Chess One of the most famous examples is IBM's Deep Blue [Campbell et al., 2002] chess program that, in 1997, was able to beat the world champion at the time, Garry Kasparov, in an exhibition match that had a lot of media buzz [Yannakakis and Togelius, 2018]. Nowadays, the best chess engine is AlphaZero [Silver et al., 2018], Google DeepMind's property, which beat in 2017 the best chess program at that time, Stockfish.

Checkers was also one of the first board games used for AI research, hence they are nicknamed the *"drosophila of AI"* by many researchers. The first attempt at solving checkers goes back to 1959 when A.Samuel developed a program to play the game [Yannakakis and Togelius, 2018]. Nonetheless, only in 1994, a computer program called Chinook [Yannakakis and Togelius, 2018] managed to beat the world champion for the first time, using alpha-beta search. By 2007, Chinook had totally solved the game [Yannakakis and Togelius, 2018].

Backgammon One game where AI achieved supremacy over human players prior to chess and checkers, was Backgammon [Yannakakis and Togelius, 2018]. In 1992, G.Tesauro created a program called TD-Gammon [Yannakakis and Togelius, 2018], combining reinforcement learning with neural networks, playing the game against itself a few million times. The program was able to perform at the same level as top human players.

Othello was also mastered by the computer [Russell and Norvig, 2010], with the introduction of Logistello [Yannakakis and Togelius, 2018]. This program beat the Othello world champion in 1997.

Scrabble In 2006, Quackle [Yannakakis and Togelius, 2018] managed to beat the Scrabble world champion.

Go More recently, in 2017, Go was the last classic board game that was finally "cracked" by AI. Google DeepMind presented AlphaGo [Silver et al., 2016], a program that beat the number one ranked Go player in the world, featuring a deep reinforcement learning approach.

Video Games AI advancements did not happen only using board games as testbeds [Yannakakis and Togelius, 2018]. As a matter of fact, in 2014, Google DeepMind [Mnih et al., 2013] also managed to create an algorithm that learned how to play several games from the Atari 2600 video game console better than any human could possibly play. The agent, which consisted in a deep Q-network, used solely the raw pixels and the game score as input. Furthermore, the real-time strategy video games *Starcraft* (Blizzard Entertainment, 1998) and *Starcraft II* (Blizzard Entertainment, 2010) went to become famous benchmarks among game AI researchers, with the existence of an already large number of algorithms to play the game, or at least, parts of it. The use of *Starcraft II* in research was also propelled by the announcement of two industry giants, Facebook and Google DeepMind, that started using the videogame for research of their own [Yannakakis and Togelius, 2018]. As a matter of fact, in 2019, DeepMind's AlphaStar agent achieved grandmaster level in *Starcraft II*, performing better than 99.8% officially ranked human players [Vinyals et al., 2019]. AlphaStar uses both supervised and reinforcement learning methods. First one is applied so the agent can learn to imitate human players. Then, reinforcement learning is used to improve AlphaStar's performance even further, utilizing games where the agent plays against itself as training data. AlphaStar's learning experience is equivalent to 60 000 years of human experience [Zhao et al., 2020].

Natural Language Processing In the field of natural language processing, IBM created a software, called Watson [Ferrucci et al., 2011], able to answer questions addressed in natural language. In 2011, the program went on to win 1million dollars in the Jeopardy TV game, being pitted against former winners of the game [Yannakakis and Togelius, 2018].

2.1.2 What Type of Games Can AI Play

AI techniques can be used to play a wide range of games, from board and card games, to complex strategy and action combat games [Lecky-Thompson, 2008] [Yannakakis and Togelius, 2018].

Board Games As previously illustrated, AI techniques can be used to create forms of agency that can play board games. Due to its deterministic nature, perfect information and narrow-scope of required skills to play, board games provided ideal environments to test some AI techniques, therefore being the first type of games used in AI research. Board games are mostly played recurring to tree-search algorithms. As a matter of fact, the tree-search algorithm Minimax was invented to play chess [Yannakakis and Togelius, 2018]. Due to its large branching factor, Go couldn't be solved with Minimax, which incentivated the creation of the Monte Carlo tree search algorithm [Yannakakis and Togelius, 2018].

Card Games In card games, the common approaches to play the game are by rule-based systems, Monte Carlo tree-search, reinforcement learning and evolutionary algorithms [Niklaus et al., 2019]. Rule-based systems are created with human knowledge of the game. Such systems are mostly constituted by condition-action statements. With reinforcement learning methods, complex games like Poker can also be played at an expert-human level. In [Zinkevich et al., 2009], a novel algorithm was proposed that advanced the state-of-art at that time, regarding playing card games, called Counterfactual Regret Minimization. Google DeepMind also developed an agent, called DeepStack [Moravčík et al., 2017], that attained expert-human performance in Poker's variant *Texas Hold'em.*

A few works also used evolutionary algorithms to play card games [Niklaus et al., 2019]. For instance, in [Mahlmann et al., 2012], an evolution algorithm is coupled with an artificial neural network to create an agent capable of playing the card game *Dominion*, although the focus of the work is on game balancing purposes. In card games with deck building as a mechanic, one possible approach is to consider the deck as the genome and the fitness function using simple agents to play the game, like in [Garcia-Sanchez et al., 2016].

Classic Arcade Games are also common playgrounds for AI. Some games like *Infinite Mario Bros* (Nintendo, 1985) and *Pac-Man* (Namco, 1980) constitute some of the most popular games used as benchmarks in AI research. The Atari games solved by DeepMind [Mnih et al., 2013] are also inserted in this category of games.

Strategy Games are a harder type of game for an AI to play and to master. These games are divided in two types: real-time and turn-based. While the hardest challenge in these games is to formulate a long plan, the first type adds an additional challenge of speed in decision-making and planning. Notable examples of strategy games are the *Starcraft* franchise (Blizzard Entertainment) (real-time), *Civilization* franchise (Firaxis Games, 2K Games) (turn-based) and *XCOM* franchise(Firaxis Games, 2K Games) (turn-based).

Simulation Games This genre includes games like *The Sims* franchise (Electronic Arts), *Crusader Kings* franchise (Paradox Interactive), *Dwarf Fortress* (Bay 12 Games, 2006), and *Black & White* (Electronic Arts, 2000). These games do not have an actual goal, and the player is only playing against himself and the environment [Lecky-Thompson, 2008]. The game test scenario proposed for this dissertation belongs to this genre.

Adventure and Exploration Games This category contains games like open-world RPGs (Role-Playing Games). Two popular examples of such games are *The Elder Scrolls* (Bethesda Softworks) franchise, and *The Witcher* (CD Projekt Red) saga. In these games, there are several activities that NPCs must be prepared to perform, such as planning actions and interacting with the player (sometimes the interaction actually corresponds to fighting the player) [Lecky-Thompson, 2008].

Motor-Racing Games Racing games are also a common type of game where AI is used to play the game. Despite at first looking an easy type of game for an artificial agent to play, several concurrent tasks are required to be executed. Breaking, steering, and gear shifting are some of the activities that the AI must be prepared to do at the same time [Lecky-Thompson, 2008]. In this genre, there are games with varying degrees of realism: from *Mario Kart* franchise (Nintendo), where the cartoonish drivers can use weapons against the opponents, to realistic driving simulators like *F1* franchise (Codemasters).

Action Combat Games First person shooters are the games that composed the bulk of the action combat genre. As Guy Lecky-Thompson says in his book [Lecky-Thompson, 2008], *"AI in action combat games has to strike a balance between perfection and incompetence, which is extremely difficult to get right."* Indeed, if the AI opponent is modeled to play optimally, boring and predictable gameplay will happen. For instance, camping (staying positioned in the same spot waiting to catch opponents off guard) is a high-performance strategy to adopt while playing first-person shooters, but said strategy is seen as unsportsmanlike and boring conduct [Yannakakis and Togelius, 2018]. Examples of first-person shooters include *Half-Life* (Valve Corporation, 1998) and *F.E.A.R.* (Monolith Productions, 2005), which are going to be mentioned in more detail in the subsection 2.1.3.

Fighting Games Examples of fighting games include *Mortal Kombat* (Warner Bros. Interactive Entertainment) and *Street Fighter* franchises (Capcom). In this type of game, the action is fast-paced and there is a relatively short amount of moves to choose from (therefore, the search space is small). A fighting AI typically is implemented as a rule-based AI and is characterized by having discrete short term goals and a reactionary mechanism to counter human player attacks. However, since the main focus of these games is on fighting the AI opponent, there are plenty of opportunities for the AI to learn from the player and adapt to his strategy [Lecky-Thompson, 2008]. Therefore, adaptive AI is one way to achieve a better experience for the player.

2.1.3 AI Methods In Games

Ad-Hoc Methods

Finite-State Machines were the most predominant technique used in the game industry, until the mid 2000's, to deal with Non-Playable Character (NPC) behaviour [Yannakakis and Togelius, 2018]. Finite-state machines are represented by graphs, in which the nodes, formally called states, commonly represent a possible behaviour of the character, and the edges, called transitions, denote the possible shift from a state to another. Each state has certain actions associated, that were *ad-hoc* designed. In other words, when the system enters a new state, pre-defined actions on entering the state are triggered. Then, when leaving the state, it might trigger two different sets of actions: a set of actions of the current state when leaving it, and the predefined set of actions on the new state when entering it.

Finite-state machines have several advantages like being very easy to design, implement and debug [Yannakakis and Togelius, 2018]. However, they present a serious disadvantage: the lack of scalability. In fact, in a large and complex project like a AAA video game, maintaining a finite-state machine might turn out unsustainable, since adding a new state to the system implies a large number of modifications and new transitions. Therefore, finite-state machines do not provide room for adaptability, and lead to a predictable pattern of behaviour.

Valve's classic first-person shooter *Half-Life*, released in 1998, held a big role in popularizing finite-state machines as the game AI main method for dealing with NPC behaviour [Thompson, 2019a, a]. The game's source code is publicly available. Examining it, one can see that every character is of a type named *"Monster"*. This core class contains a definition of a "state", albeit it does not correspond to a state of a finite-state machine. Instead, it describes what the character is currently doing. Each subtype of class *"Monster"* also contains specific tasks that each instance of that type can execute. In total, there are eighty unique tasks in the *Half-Life*'s codebase, and these tasks correspond to states of the finite-state machine. In addition, each character has conditions, which represent the current information that it possesses, along with sensors that allow the update of the conditions (there are sensors of vision, sound, and smell). These conditions correspond to the transitions of the finite-state machine.

One variant of the original finite-state machine is the hierarchical finite-state machine [Thompson, 2019a, a]. This type of system helps to deal with a larger number of states and transitions, since it allows the transition not only to a particular state as in normal finite-state machines, but also to a group of several states. Therefore, it can also provide a way to transition from one finite-state machine to another. Hierarchical finite-state machines still have been used as recently as 2016 [Thompson, 2019a, a], in video games like *Wolfenstein: New Order* (Machine Games, 2014) and the reboot of *DOOM* (Bethesda Softworks, 2016).

Another variant is the fuzzy finite-state machine, and it is a way to avoid the predictable pattern of behaviours of the normal version of the system [Lecky-Thompson, 2008]. A fuzzy finite-state machine can transition to multiple states and, consequently, can be in several states at the same time. There are two subcategories of fuzzy finite-state machines: fuzzy state FSM and fuzzy transition FSM [Lecky-Thompson, 2008]. In the former, a "state" can actually be a set of several states in which the system is, and with the possibility of being in them at different degrees. For instance, to simulate driving a vehicle, a fuzzy state FSM can be applied, since one can be turning, braking and shifting gears at the same time. Each task is therefore a state, and the system can be in several states at the same time, with varying degrees of membership. The second subcategory, the fuzzy transition FSM, simply has assigned probabilities to each transition, with the possibility of each state having several possible transitions. Only one transition can be followed at a time, as in the normal finite-state machine, but that might still lead to a different behaviour when transitioning from one state to another, since there are multiple transitions to choose from.

Behaviour Trees are directed acyclic graphs [Thompson, 2019b, b] that, similarly to finite-state machines, model transitions between a set of behaviours. Their advantage over finite-state machines is the modularity of the former. In fact, they can generate complex behaviours composed of multiple simple tasks [Yannakakis and Togelius, 2018]. Like finite-state machines, behaviour trees are easy to design, implement and debug, with the main difference between the two methods: one being composed of states and the other made of behaviours [Yannakakis and Togelius, 2018].

A behaviour tree starts its execution in the root node, and goes down the tree in a sequential manner to execute behaviours, that are in the leaf nodes. Along the way, there are other three types of nodes: selector, sequence and decorator. The first one, the selector node, decides which child node to execute based on game world information. A selector node can be, more specifically, a priority selector or a probability selector. In the former, child nodes are ordered by priority and tried one after the other until one child node is executed successfully.. In the latter, each child has a probability to be executed, *ad-hoc* specified by the designer. The second type of node, the sequence node, allows the execution of several child nodes in succession. The last one, the decorator node, provides more complexity for a single child node. For instance, it might allow repeating several actions or even make a selector to make the opposite decision of what it should choose given the available information.

Behaviour trees are the cornerstone of modern game AI, being the most dominant method in the game industry, after its successful implementation in *Halo 2* and *Bioshock* [Yannakakis and Togelius, 2018].

Utility-Based AI consists of the agent having an internal performance measure, called utility function, that calculates the importance of different aspects of the current game state. The aim of the agent is to maximize its utility function [Russell and Norvig, 2010].

One example of the use of utility-based AI can be seen in the modern action RPG *Horizon Zero Dawn* (Guerrilla Studios, 2017), where the player has to fight electronic creatures that have resemblance with biological animals. In this video game, an utility function is used to manage AI behaviour of the machines when entering combat against the player [Beij, 2017]. More specifically, the utility function is used to assign combat roles to each enemy machine when the player is spotted by any member of the herd. For each machine, its utility is calculated for each available role, and the one who scores highest will get that role assigned to it. For instance, for the *"attacker"* role, the closer to the player a machine is, the greater its utility value will be for that role.

Utility-based AI has several advantages over other ad-hoc methods [Yannakakis and Togelius, 2018]. First, it presents a greater amount of modularity, since the utility function depends on several variables, and this list of factors can be dynamic. Second, it has room for improvement and change, since adding new variables to have in consideration in the utility function is easier. Finally, the method is also reusable, since it can be extrapolated not only from a decision to another, but even from a game to another game.

Utility-based agents are also superior to goal-based agents (approached next), in two distinct cases [Russell and Norvig, 2010]: when there are goals that conflict with each other and when there are several goals possible to be reached, but none can be reached with certainty. In the former case, the utility function can decide the appropriate trade-off and choose which goal is the best given the current situation. In the latter, the utility function allows to decide between one of the several possible goals by having in consideration the relationship between the probability of success and the importance of objectives.

However, Utility-based AI also has some limitations. As any other *ad-hoc* method, the agent will never be smarter than the game designer who specifies the utility function [Rasmussen, 2016]. Also, it requires time for testing, to tweak the utilities in order to ensure correct and smart behaviours [Rasmussen, 2016]. In addition to that, it is not always easy to formulate an appropriate utility function, and to properly justify where the numbers come from [Wooldridge, 2005].

Planning-Based Approaches

Goal-Oriented Action Planning Roughly at the same time of the rise to popularity of behavior trees, another novelty method gained traction in the game industry, Goal-Oriented Action Planning (GOAP)[Orkin, 2003]. This method was first implemented with success in the video game *First Encounter Assault Recon* (or *F.E.A.R.* for short) (Monolith Productions, 2005) [Orkin, 2006], a survival horror first-person shooter.

GOAP resembles STRIPS (short for Stanford Research Institute Problem Solver) [Fikes and Nilsson, 1971], a planning system from academia. In STRIPS, a particular desired game state is known as a goal. A goal can be achieved through a sequence of actions. An action has preconditions (conditions that are required to be satisfied so the action can take place) and effects (the changes that the action causes in the game world when done).

Therefore, planning consists in calculating a path, preferably the most efficient, from the current game state to the goal-state, by searching for a sequence of actions that can lead to the desired state. Portions of the original source code of F.EA.R are available publicly [Thompson, 2020], which allows one to observe that there are over 70 goals and 120 actions in the codebase.

GOAP differs from STRIPS in four ways [Orkin, 2006]:

- GOAP has a cost per action, so if there are two valid plans to satisfy the same goal, the most efficient one (the one which the sum of actions cost is lower) is chosen over the other. GOAP uses A* algorithm to do a reverse search from the goal state to the current state of the game world.
- Effects are specified by fixed-size arrays that represent the world-state, instead of using the Add and Delete Lists of STRIPS (STRIPS uses lists to add and remove knowledge about the world).
- It has procedural preconditions. In other words, checking on-demand if an action is possible to execute.
- It has procedural effects. After the planner devises an executable plan, its actions are "activated", which means the planner is going to transmit to the finite-state machine in which the character must be, in order to execute the next action of the plan.

F.EA.R. also uses a finite-state machine, but a very simple one, with only three states: a state for the character to go to a location, a state for playing a certain animation, and another state for interacting with an object. The in-game objects tell the character what they can do interacting with it, therefore being called *"smart objects"*. This technique is also used in *The Sims* (Electronic Arts,) and *SimCity* (Electronic Arts) franchises, with Will Wright, creator of those games, calling the method *"smart terrain"* [Lecky-Thompson, 2008]. This is used mostly for game performance, since usually there are not enough resources for the characters to learn every possible action available in-game.

The main advantage of GOAP, over methods like finite-state machines, is that it allows the decoupling of goals and actions [Orkin, 2006]. In fact, while in finite-state machines, each state is intertwined with the others over transitions. Therefore, adding a new state can be quite cumbersome. Meanwhile, in GOAP, one can just create a new action with the respective preconditions and effects, knowing that the planner will dynamically link the new action with any goal that is satisfied by the action effects.

Hierarchical Task Network Planning However, planners like GOAP and STRIPS are being adopted less frequently than before, in favor of another planning method, called Hierarchical Task Network Planning (HTN Planning) [Thompson, 2020]. HTN Planning generates plans composed by high-level actions. These abstract actions are then decomposed in more low-level concrete actions. With this composition of smaller actions into a larger higher-level one, the creation of more complex plans is possible [Beij, 2017]. HTN Planning has been applied in modern AAA video games like *Dying Light*, *Max Payne 3* and *Horizon Zero Dawn* [Thompson, 2020].

Tree Search This family of algorithms consists in building a tree where nodes represent possible game states and edges symbolizes possible actions that the player can take to go from one state to another. The initial node of the tree is the root, and represents the game state from where the search procedure starts. This game state is, most of the time, the current state of the game. There are several tree-search algorithms, each one with multiple variants, to be able to adapt to the particular game where it is being applied [Yannakakis and Togelius, 2018].

Tree search algorithms can be divided in two sub-categories, classic and stochastic tree search [Yannakakis and Togelius, 2018]. In classic tree search, algorithms feature no randomness and are the ones who have been used since the beginning of research about AI in games. However, more recently, stochastic tree-search came into the scene, with the creation of the famous Monte Carlo Tree Search (MCTS) algorithm to play the board game Go [Yannakakis and Togelius, 2018].

Regarding classic tree search, the most used type of algorithm is best-first search. In particular, A* search and its multiple variations are the most common methods for pathfinding in video games. As seen in the description of GOAP algorithm, used in F.E.A.R. video game, A* can be used for planning too [Orkin, 2006]. The nodes of the graph are world states and the edges represent the possible actions. Classic tree search performs well in deterministic single-player games with perfect information and a low branching factor.

The same cannot be said regarding its performance in adversarial games, since there is another player that is trying to win the game, and therefore, the actions of a player might depend on the actions of the other. To deal with this, there is Minimax, the basic adversarial search algorithm [Yannakakis and Togelius, 2018].

In games with hidden information, like card games, instead of Minimax, the common method used is Monte Carlo Tree Search [Yannakakis and Togelius, 2018]. This algorithm introduces randomness to deal with hidden states and high branching factors, in the form of rollouts. This consists in randomly selecting actions from the current selected game state until a terminal state is reached.

In the videogame industry, Monte Carlo tree search has seen successful implementations in the likes of the real-time strategy Total War franchise [Thompson, 2018].

Supervised Learning

In supervised learning, the goal is to map a set of inputs, named features, to a set of outputs, named targets, based on previous examples of pairs (feature-target). According to the type of the output, supervised learning methods can be divided in three groups [Yannakakis and Togelius, 2018]:

- Classification: used when output is represented by a nominal set of classes;
- Regression: used when output is defined by a numeric value or interval of values;
- Preference learning: used when output is of ordinal type, like rankings and preferences.

Common methods used for supervised learning in games include artificial neural networks, support vector machines and decision tree learning [Yannakakis and Togelius, 2018]. The usual way for an agent to play a game via supervised learning is to make it learn from previous examples of human player behaviour, in which the features correspond to game states and the targets represent the action the human player took in those game states.

Reinforcement Learning

In reinforcement learning (RL), when the agent chooses an action a that makes it transition from the original state s to the new state s', it receives a reward [Russell and Norvig, 2010]. Therefore, in this machine learning method, the agent's goal is to unveil a policy that selects the actions that maximize the total reward obtained from said set of actions. This configuration commonly leads to a classical problem of RL called exploration vs exploitation [Russell and Norvig, 2010]. In fact, the agent must make a tradeoff between exploring unknown states and exploiting the already known rewards [Russell and Norvig, 2010].

With a discrete notion of time, at a given timestep *t*, the interactions with the world can be represented as a Markov Decision Process (MDP). A MDP is defined by the following features [Russell and Norvig, 2010]:

- *S*: a set of possible world states;
- *A*: a set of possible actions;
- *P(s, s',a)*: a transition model that specifies the probability of going from state *s* to state *s'* by executing action *a*;
- *R(s, s', a)*: a reward function that stipulates the reward received from transitioning from state *s* to state *s'* by opting for action *a*.

Two of the most important subclasses of Reinforcement Learning (RL) are Temporal-Difference Learning and Evolutionary RL[Yannakakis and Togelius, 2018].

Temporal-Difference Learning This class of methods is model-free because it does not require knowledge of the world beforehand. Instead, it uses estimates of previous experience to update its values. This notion of estimating based on an existing estimate is called bootstrapping [Yannakakis and Togelius, 2018]. TD learning also uses the notion of backup, which consists of reverting from the future state that is being explored back to the current state. [Yannakakis and Togelius, 2018].

Evolutionary Reinforcement Learning In this class, there are two methods: neuroevolution and genetic programming [Yannakakis and Togelius, 2018]. In the former, the key idea is to use evolutionary algorithms to evolve the weights and topology of neural networks. In the latter, evolutionary algorithms are used to evolve the nodes of a tree.

Reinforcement Learning is starting to gain traction in the game industry as seen in [Jacquier, 2019] and [Li, 2020].

Dynamic Scripting

In [Spronck et al., 2004], a hybrid method called dynamic scripting is proposed. Dynamic scripting combines reinforcement learning with rule-based Artificial Intelligence (AI). There is a rulebase for each type of character, that contains rules that dictate the character's behavior. Each rule has an associated weight value that defines that rule's probability to be inserted in the controller script of a newly generated character. The weight values are updated at the end of an encounter (a confrontation between the player and the AI character) via reinforcement learning techniques. The weight update is based on the contribution of each rule to the overall performance of the character in that particular encounter.

In order to assess the practical use of dynamic scripting in games, the method was tested in the commercial *Neverwinter Nights* RPG (BioWare, 2002). Results show that dynamic scripting is an efficient way towards adaptive AI, since it can reliably learn from previous encounters with the player and change its tactics accordingly.

However, one limitation of this method is the static nature of the rulebase. Despite the associated weights evolving, the rules themselves remain the same. This leads to a finite amount of possible combinations of rules.

2.1.4 AI Roles In Games

When playing the game, AI can portray the role of either a player or a non-player character [Yannakakis and Togelius, 2018]. Regardless of the game role AI is occupying, it can have one of two distinct goals within the game. Those goals are playing to win or playing to enhance the human player experience [Yannakakis and Togelius, 2018]. Playing to win means exactly that: the AI performs as optimally as possible to beat the game. However, an AI that presents optimal performance to be victorious every single time might culminate in predictable and boring strategies, undermining the human player experience (like *"camping"* in first-person shooters). It is also worth noting that some games do not even have a clearly defined "win state" like *Minecraft* (Mojang Studios, 20011) and *Crusader Kings* franchise (Paradox Interactive). Therefore, AI can assume four distinct roles depending on these two factors, game role and goal [Yannakakis and Togelius, 2018].

Playing to Win as a Player

An AI playing the game to win in a player role is the most common scenario in academia, where games are used as benchmarks for the AI itself [Yannakakis and Togelius, 2018]. Reasons to use games as testbeds for AI were already listed previously, in the beginning of this section.

Besides AI benchmarking, another reason to use AI in the player role to win the game is when there is the need for an opponent that provides an appropriate challenge to the human player, like in games of perfect information [Yannakakis and Togelius, 2018]. However, in imperfect information games, allowing the AI to "cheat" is an easy way to make it more challenging to the player. This can be done by providing information to the AI that is otherwise unknown to the human player, for instance, by accessing the full state of the game. For instance, in the strategy video games *Civilization* saga (Firaxis Games), there is not a single AI until today that can play the game at the same level of a human player, without cheating.

Playing to Win as a NPC

There are games, unlike the *Civilization* franchise stated previously, in which the human player can't portray a certain game role. For instance, in turn-based strategy *XCOM* video games franchise (Firaxis Games), the player has to control a squad of characters that have to defend Earth from aliens. Fighting these enemies is the main core gameplay mechanic of the game, so it would not make sense if the player was able to control the alien side. Therefore, in this particular scenario, there is the need of an AI that despite being in a NPC role, its aim is to win the game. In this concrete game case, the goal of the aliens is to defeat the player.

Sometimes, developing a NPC to win is a precursor of creating a NPC to enhance human experience [Yannakakis and Togelius, 2018]. In fact, this situation arises when one wants to implement "rubberband AI", which means tweaking the performance of the AI so it will always provide a decent challenge to the player, while exhibiting variable behaviour. This scenario is very common in racing games, where one wants to make sure that the AI racers won't ever fall too far behind nor stay too ahead of the player, because either situation would result in a boring or frustrating gameplay. Thus, to be able to implement these performance boundaries, the NPC has to know how to play to win the game in the first place.

Playing to Enhance Human Player Experience as a Player

AI can also play the game in the player role not to win the game, but for human player experience [Yannakakis and Togelius, 2018]. However, one could pose the question: if the AI is the player, where and how does human player experience come into the picture? In fact, this configuration is mostly used when one wants to attempt simulation-based testing [Yannakakis and Togelius, 2018]. In this type of testing, an agent that mimics the behaviors of a human player is required, otherwise the test feedback might be untrustworthy. For instance, when applying simulation-based testing to the evaluation of a game's new level, the agent can wrongly report that the level is playable, but only with extremely fast and inhuman reflexes. Thus, the evaluator agent must act like a human player would be able to act in the same situation. That implies having similar reaction speeds, making the similar mistakes that a human would (which theAI agent would not do if following an optimal strategy), taking unnecessary risks, going for serendipity, and even imitating the innate curiosity by exploring unknown things in the environment [Yannakakis and Togelius, 2018].

Another reason to use AI in this scenario is for game demonstration purposes [Yannakakis and Togelius, 2018]. A lot of games have a common feature that is a demo mode, which involves demonstrating to the player how to play the game. In the case that all the content is known *à priori*, such demonstrations can be hardcoded. However, in games with procedural generated content, or with content designed by the player itself (such as in-game level-editors), said demonstrations have to be generated by the game [Yannakakis and Togelius, 2018].

Playing to Enhance Human Experience as a NPC

An AI in a NPC role, with the aim of enhancing human player experience is by far the most common scenario in the game industry. Most commercial games put a large amount of emphasis on their NPCs, to make the game world feel more immersive to the player. They can have a wide range of in-game roles, like cooperating with the player, providing the player with quests, telling a story or just being in the background doing their daily routines.

Modern videogames are highly complex in terms of graphics and game environment, with most of the processing power being dedicated to rendering the world. Therefore, most games cannot actually implement complex AI for every NPC inhabiting the world. So, most of the time, what the game designer wants to achieve for the NPC is the "illusion of intelligence", rather than actual intelligence [Yannakakis and Togelius, 2018]. Illusion of intelligence consists in making the player believe that the NPCs are actually smart, despite them being controlled by simple scripts. Actually, one can even argue if most scripts used in the game industry to control NPCs are considered AI, due to the sheer simplicity of them [Yannakakis and Togelius, 2018].

A NPC presenting likeness to human behaviour, or not, depends on what the game designer wants the NPC to represent. For instance, if the NPC is portraying a non-human creature, like a monster, it shouldn't resemble human-like behaviour. In other instances, one desired characteristic of a NPC is predictability. Indeed, stealth-based games revolve around predictable patrol patterns that enemies take while patrolling, so the player can devise the best approach to the situation. And in games with "boss battles" (a really strong NPC that the player has to defeat to progress in the game), the player has to also memorize the patterns of attacks and movements of the NPC, so he can find the precise moment to strike on the opponent's weak points.

AI Roles Conclusion

One of the main differences between academia and the game industry is the emphasis they place on one of the four types of scenarios described above regarding AI roles in playing games. In fact, in academia, using games as benchmarks for the AI itself is the norm. On the other hand, industry focuses on using AI as a way to improve player experience, most commonly in the form of believable NPC's. This discrepancy of mindset only aggravates what is known as the "gap" between academia and the game industry, regarding both techniques used and knowledge possessed. Nowadays deep learning methods are popular among academic researchers. However, most game development companies prefer to stick with software engineering methods like finite-state machines and behaviour trees that, although they increase authorial burden, are well-proven methods. The main reasons for this industry's resentment towards using academy methods are that machine learning techniques reduce the control of the game designer/developer over the results, making it hard to test and debug the game, and that these methods fall short regarding scalability for massive scale projects, like *AAA* (high budget) video games.

However, this gap between academia and industry has been decreasing over time, and there are already successful cases of game development companies incorporating machine learning techniques in their development process. For instance, *NetEase Fuxi AI Lab* was created in September 2017 with the goal of *"enlighten games with AI"* [Li, 2020]. They created their own reinforcement learning framework, RLEase, and have been applying it in games which are property of NetEase Games, like *Justice Online* and *Fever Basketball*, to produce more challenging NPCs.

Finally, there are also common efforts to reduce the gap even more. One particular case is Ubisoft La Forge [Jacquier, 2019], a partnership between researchers and Ubisoft, an *AAA* game development company. This collaboration consists in Ubisoft granting researchers access to their equipment and more importantly, their privata data, to propel faster advancements of AI techniques that can be used within Ubisoft games [Jacquier, 2019]. This way, the partnership is beneficial for both parties. For Ubisoft, they get first-hand access to AI advancements that can be incorporated in their games. For academics, one of their main issues is mitigated, that is the lack of access to large amounts of data.

The AI implemented on this thesis' work assumes the role of playing to enhance human experience either as a player and as a NPC. Thus, simulation-based testing will be a focal point of this chapter, which in turn creates the need to model believable agents. Therefore, the next section will dwell deeper into the field of simulation-based testing, with the section after that being about believable agents and how to model them.

2.2 Simulation-Based Testing

In this section, previous work regarding simulation-based testing is approached. In the end of the section, a quick detour is made to talk about a particular simulation scenario: disease modeling.

The continuous works of Holmgard et al. demonstrate the usefulness of using agents as playtesters of generated content [Holmgård et al., 2014, a][Holmgard et al., 2014, b][Liapis et al., 2015] [Holmgård et al., 2016] [Holmgård et al., 2019]. Using simple turn-based rogue-like games as testbeds, Holmgard et al. define several playstyles archetypes. For instance, the Runner focuses on completing the level with doing as few moves as possible while the Treasure Collector consists in collecting the maximum treasure objects possible.

First, in [Holmgård et al., 2014, a], Q-learning is used to model agents, so their decisions can be compared to ones of human players. Despite the limitations of the used method, authors conclude that the idea of *procedural personas* has potential.

In [Holmgard et al., 2014, b], seven linear perceptrons are combined with an evolvable controller to model the agents, and these are compared to the previous Q-learning agents modeled in [generative agents for decision making], as well as human players. The evolutionary algorithm used is a truncation-based one that keeps the best half of the population and replaces the worst half with offsprings from the remaining individuals. Authors conclude that the new modelled agents take the same action of human players between 60% and 88% of the time. This large variation is due to differences between game levels. Therefore, it presents better performance than the previous Q-learning agents.

In [Liapis et al., 2015], a feasible-infeasible two-population evolutionary algorithm is used to evolve levels of the game used as testbed. Then, the *procedural personas* built in previous works are used to assess both playability and the quality of the evolved levels. This assessment is utility-based. For playability, it takes into consideration if the generated level has a specific amount of tile types (for instance, one entrance tile and one exit tile), if the agent can finish the level without dying by maximizing its utility, and if all the game objects in the level can be reachable by the agent. For level quality, the evaluation is dependent on the type of *persona* testing the level. For instance, for the "Treasure Collector" *persona* (its main goal is to collect as many treasures as possible), level quality is dependent on the number of treasures in the level. However, one must also consider the meaningful decisions of the agent. After all, in the example regarding the Treasure Collector *persona*, if the evaluation of level quality is solely based on the number of treasures in the level, nothing prevents from generating a level with a treasure in every tile between the entrance and the exit of the level. Scenarios like that one are uninteresting, since the player does not make any meaningful decisions throughout the level. So, a level's risk involved in the agents decisions is also considered in the evaluation process.

The author shows that the obtained results are an important demonstration of not only the potential of the evolutionary algorithm used to generate levels, but, more importantly, the reliable use of *procedural personas* as evaluators of the generated content.

In [Holmgård et al., 2016], the goal is to use once again the previous *procedural personas*, and this time, compare the *personas* with clones, that are agents created to reproduce human play traces (see subsection 2.3.4 for a definition of play traces). Two metrics are used to compare the performance of both types of agents. One is the action agreement ratio, that considers the actions taken at each play trace. If the agent takes the same decision as the human, it is awarded one point. Then, the agent's total of points is divided by the number of total actions it had to make in the level to compute the action agreement ratio. The second metric is the tactical agreement ratio. It works in a similar way to the first ratio, but it only considers interactions with game objects (killing a monster, collecting a treasure, etc).

The authors conclude that although *personas* and clones perform almost equally at action level, clones present better results when game object interactions are taken into account.

[Holmgård et al., 2019] builds upon the previous works by modeling the agents using the MCTS method and genetic programming. Instead of the Upper Confidence Bound (UCB) formula used to select a node in the selection phase of the algorithm, it was replaced by genetic programming . The formula is given by a syntax tree where the nodes are binary operations and the leafs are constants or variables, after 100 generations of evolution. Rollout to a terminal state can last a large amount of time, so instead the authors decided to simulate just the ten next moves randomly and then backpropagate the utility score of the state reached with those ten moves. The utility function is obviously different for every playstyle archetype. For instance, for a persona with the objective of finishing the game the fastest it can, its utility function will focus on its proximity to exit and the number of steps taken. If instead it is a persona who has the priority of collecting as many game items as possible, it will take into account the number of game objects possible to interact with, while a parameter like steps taken is irrelevant.

The experiments done show that the evolved MCTS agents are able to play the game faster than the baseline MCTS agents (with UCB formula) while still being different from each other (the initial archetypes are not lost).

The key thing to note about these works is that the *personas* (the "archetypes" of playstyles) are designed *à priori*. Therefore, the *procedural personas* method is only useful if the game designer can find initial utility functions. One solution proposed (but not implemented) in [Holmgård et al., 2019] to overcome this limitation is to learn utility functions from actual human players.

More recently, [Zhao et al., 2020] shows two ways of how agents can be used in playtesting a game. The work first considers *Sims Mobile* (Electronic Arts) as one of its case studies. *Sims Mobile* is a mobile game that simulates life and there is no specific goal to be achieved by the player. The aim is to optimize player experience. The authors use A* algorithm as a planning technique to develop agents able to play the game. They validate the method by comparing it with evolution strategies, where the agent plays to optimize a utility function that selects an action from all the available ones. The obtained results show that the A* algorithm manages to acquire equivalent results to the evolution strategy, which is a computationally more expensive algorithm to run than A*. In another case study, the goal is to build an agent to play a mobile game like an expert player would, to measure expert player progression. For that, deep reinforcement learning is used. More specifically, both a DQN (Deep Q-Network) agent [Mnih et al., 2015], which is a combination of Q-learning algorithm with convolutional neural networks, and a Rainbow agent [Hessel et al., 2018], which contains several extensions to DQN, are used.

The results show that both types of agents achieve a reasonable amount of success, although training time is still very costly and other options are required to speed up the process.

2.3 Believable Agents

As seen in the previous section, there are three main reasons for the need of believable agents: for simulation-based testing, for game demos, and arguably the most important one, to enhance human player experience. Therefore, this section first describes what believable agents are, and how can someone measure believability. Then, it follows with the main approaches used today in believable agent modeling, listing several case studies as examples. Finally, the section culminates with the listing of several other AI areas that can influence believable agents research.

"Believable agent" is a term coined in the 90's to define agents that present natural behaviour. It first appeared in works of J.Bates [Bates, 1994], member of the Oz project. The interest in such agents grew exponentially with the rise of the game industry, since, despite no formal proof, it is generally believed that human-like characters add a lot to the entertainment value of a game. This is indeed the main motivation to create such agents, and it is something that the game industry has been giving an egregious amount of focus. Other reasons for the existence of said characters are, as mentioned in the previous section, for simulation-based testing and game demos. In the former, believable agents are required when one wants to test the simulation scenario and obtain feedback of what a human player would probably do in that scenario.

2.3.1 Requirements

But how does one build and measure such a subjective structure? First, it is necessary to look at the requirements of an agent that is intended to behave "human-like".

In his P.H.D [Loyall, 1997], A.Loyall lists several key components that an agent has to possess to resemble human-like behaviour:

- Personality, that is the set of particular details that defines the character or agent. These details can be of behaviour, thought, or emotion.
- Emotions, that the agent has to be able to express in some way, like in its reactions to the stimuli of the environment.
- Goals, because an agent who just purely reacts to the environment lacks depth of character and feels less "alive". Also, being able to execute parallel actions while concurrently pursuing goals is another key aspect to promote convincing behaviour.
- Change, since one of the most interesting things in a narrative is to witness the character's growth, according to his characteristics [Campbell, 1991].
- Social Relationships and Context, by which it means the agent must be fully integrated with its world, having assimilated culture, social conventions, and having forged connections with other agents inhabitants of the world.
- Consistency of Expression, that is all ways the agent possesses to express itself being completely in sync with each other, just like an actor uses not only utters his lines, but also expresses the portrayed character by voice intonation, body gestures and facial expression.
- Reactiveness and Responsiveness, because human-like behaviour is hard to convey if the agent does not react to the environment, like a car that is about to hit it.
- Resource-Bounded, because if one wants an agent that resembles a human, it has to have the same capabilities and limitations. That means no inhuman reflexes and no super-intelligence (compared to humans).
- Well Integrated. By that, one means that despite the agent being composed of

several distinct modules, they all have to work like if they were merged together. Not fulfilling this requirement may lead to scenarios like an agent that stops moving just to think about what it is going to do next (i.e, the movement module stops for the planning module to take place).

All of these requirements help to convey the illusion of life to the observer, which contributes to “suspension of disbelief”. Suspension of disbelief is the simultaneous belief in two inconsistent matters. For instance, while watching a movie or a tv show, one chooses to believe the interactions between the characters. But at the same time,we are aware that we are just watching the actors speaking and moving, in a film set, according to a predetermined script. Therefore, it’s the film director’s job to make it as immersive as possible, suppressing the second idea, of actors just doing their job, in the viewer’s mind.

A similar immersion can apply in video games. If the NPCs of a game behave “strangely” the player’s suspension of disbelief breaks down, subsequently undermining the player’s game experience. Henceforth, game characters’ believability is a crucial aspect of video games, and that’s why there is a whole area of research dedicated to believable agents.

2.3.2 Assessment of Believability

At this point, it is important to make a formal distinction between character and player believability. Character believability refers to the character itself, and to the fact that the observer believes that the character fits with the narrative context. At the moment, believability based on the idea of realism is reserved to high-budget cinematography productions with abundance of Computer-Generated Imagery (CGI) effects. On the other hand, player believability takes into consideration that the observer knows that the character is not real, being just a graphical representation, but the observer believes that the entity controlling the character is real.

Since player believability is closely related with player experience, one can attempt to measure the former with the same tools as the latter [Togelius et al., 2012]. Player experience can be measured in three distinct ways: via manual annotation (either self-reporting, interviewing or direct observation), by tracking a player’s physiological data and body gestures during play, and finally, through logging data. The first method is the one used in most Turing test-based assessments, like in the 2KBot Competition [Hingston, 2009] and Mario AI Championship [Shaker et al., 2013].

Indeed, the two last aforementioned competitions, 2KBot Competition and Mario AI Championship, portrayed an important role towards efforts to build believable agents. In 2KBot Competition [Hingston, 2009], organized by Phil Hingston, participants were invited to develop agents that can play the first-person shooter *Unreal Tournament 2004* (Epic Games, 2004) in a human-like manner. The main goal of the competition is to deceive the human judges into thinking that the agent they are seeing is controlled by a human player. It was held from 2008 to 2013. Similarly, the Mario AI Championship [Shaker et al., 2013] [Yannakakis and Togelius, 2018], held from 2008 to 2012, encourages participants to develop agents able to play Infinite Mario Bros, also in a human-like manner. From these competitions, the authors conclude that agents cannot pretend to be humans just yet. Some obvious traces of non-humanness include unnaturally fast reactions, not stopping to think nor hesitating, and the absence of unnecessary actions (for instance, one common trait of most players, regardless of the game being played, is to jump while running) [Yannakakis and Togelius, 2018].

2.3.3 Top Down Modeling Approach: Social Sciences Frameworks

The approaches to build believable agents architecture can be classified as top-down and bottom-up [Yannakakis and Togelius, 2015]. In the former, architectures are built based on existing frameworks from social sciences like psychology. The latter is done through human player's imitation.

BDI

BDI (Beliefs-Desires-Intentions) is an agent architecture for practical-reasoning agents [Bratman et al., 1988]. A classic BDI agent presents three main components:

- Believes, that represent the knowledge of the world that the agent possesses.
- Desires, that define the motivations and goals of the agent.
- Intentions, that depict the goals that the agent is currently pursuing.

The core algorithm's loop goes as follows: agent perceives its environment, and updates its knowledge about the world (its Believes). Then it calculates possible goals to be achieved according to the current configuration of the world, and how to achieve them. Finally, the agent executes the plan. While performing the sequence of actions that compose the plan, the agent always checks if the plan is still sound and if the next action is still possible to be executed. Throughout the years, several works proposed modifications to BDI's original architecture, even going as far as incorporating an emotion mechanism in the algorithm's original loop [Jiang et al., 2007].

Talk of The Town and Bad News

A particular work where the main focus is to simulate a society of believable agents is Talk of The Town [Ryan, 2018]. James Ryan's et al work consists of a game-engine that simulates the everyday life of a typical american town, and a game, called Bad News [Ryan et al., 2016], where the player interacts with the characters of the town. The main focus of the game is on knowledge propagation, since the characters of the simulation spread their knowledge (that may be true or false) about the world to other characters [Ryan et al., 2015].

In order to provide a rich social network right from the start of the gameplay, a simulation that generates a century of history is made beforehand [Ryan, 2018]. Although Talk of The Town models time as day and night timesteps at the start of gameplay, the simulation to generate previous history implements level-of-detail modulation [Ryan, 2018]. This technique consists in adjusting the granularity of the simulation, mostly for the system to be more computationally efficient. Therefore, in Bad News, four days of a year are simulated and then the results are extrapolated to a different time scale, according to the time passed since the last simulated timestep [Ryan, 2018]. This process goes on for every year before the actual gameplay starts.

To embed the inhabitants of the game world with personality, the author applies to the characters one of the main psychology models, the Five-Factor Model [Mccrae and John, 1992] [Ryan, 2018], that assumes that a person is characterized by the following five factors:

- Openness to Experience, that states how creative and adventurous the person is;
- Conscientiousness, that relates with the diligence and planning skills of a person;
- Extroversion, that mostly relates with the sociability of the person;

- Agreeableness, that describes how likeable and sympathetic the person is;
- Neuroticism, that describes emotional stability. In other words, it dictates how anxious and unstable the person is.

One particular attribute that Talk of The Town's characters possess that grants them an extra layer of believability is memory fallibility [Ryan, 2018]. In fact, each character has a float type attribute for its memory, that decreases over time. The lower the attribute, more the chance of the character's knowledge being degraded.

To represent each character's knowledge, mental models are used [Ryan, 2018]. To be more precise, a character formulates and updates a mental model about other characters and locations that it encounters throughout its lifetime. Mental models can be linked with each other: for instance, if Character A sees Character B working at Workplace X, A will update its mental model of both B and X. In A's model of B, one attribute is B's occupation, that it will link with A's mental model of Workplace X. The inverse also happens: a mental model of a business contains a list of employees, so when A updates its model of X, in that list will be a reference to A's mental model of B.

Dynemotion and Mind Module

In [Eladhari and Sellers, 2008], two models to represent believable agents are proposed. The common denominator in both of them is the use of "mood", which the authors define as "the overall state or quality of feeling at a particular time" [Eladhari and Sellers, 2008].

In the first one, Dynemotion People Engine (DPE), characters have personalities, desires, goals, and emotions. Personalities are based on the Five-Factor model, while emotions are defined as "all cognitively apprehended aspects of physiological or qualitative psychological states and processes" [Eladhari and Sellers, 2008]. The mood is displayed in a bidimensional grid, in which x-axis represents "Outlook", that represents the tendency of the character at any given moment to feel good or bad emotions, and the y-axis represents "Affect", which is the intensity of the mood.

The second model described in the article is the Mind Module. In this particular model, the character's personality is given by a collection of traits also inspired by the Five-Factor Model. The model itself is represented by a weighted network of nodes, that can be of four types: mood nodes, emotion nodes, trait nodes and sentiment nodes. The mood representation is similar to DPE's one, since it depicts mood as a two bidimensional matrix where the axis also represents "Outlook" and "Affect". Finally, the model contains the notion of emotional attachments, called sentiments. One example of a sentiment provided in the article is that, if a character has arachnophobia, it would have the emotion Fear associated with the object Spider. Sentiments can also be created at runtime, which the author refers to as *"emergent sentiments"*.

Orphib II

In [Barreto et al., 2014], an agent architecture for a multiagent system that contains alien-like creatures is presented. The key idea of the model is genetic personalities, i.e., inheritance of personality from the agent's parents. In the agent's genome, its eye color is also encoded, which combined with the world's light at any given moment, affects the agent's sensors.

Results show that the population tends to increase non-linearly, as the result of a compul-

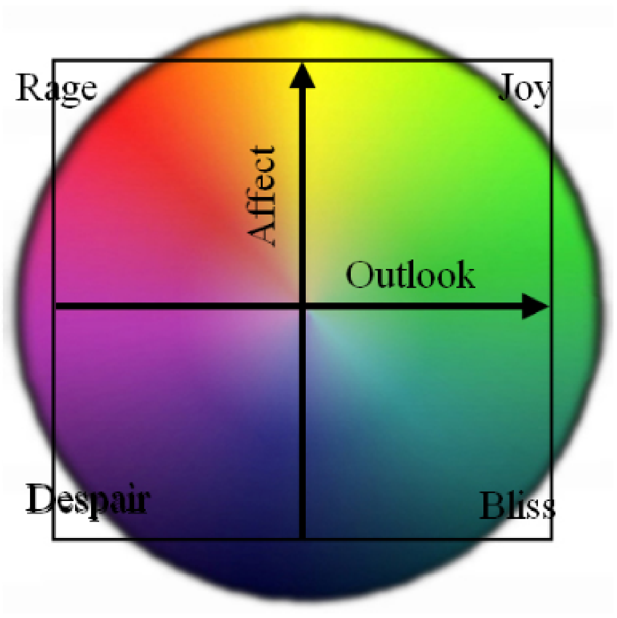

Figure 2.1: Dynemotion Mood Representation. Image taken from [Eladhari and Sellers, 2008]

sive need for reproduction of the agents.

Believability of Non-Anthropomorphic Characters

However, all of what is described above refers to agents that aim at portraying a human-like character. But what about an agent that intends to portray the role of any other creature? In [Barreto et al., 2017], a creature believability scale is proposed to assess the behaviours of zoomorphic creatures. The scale is composed of four main categories, that are:

- Relation with the environment, that encompasses both the creature's reactions and actions in the environment;
- Biological/Social Plausibility, that describes the creature's interactions with other creatures, as well as its autonomy and reactiveness to its surroundings;
- Adaptation, that means learning behaviors;
- Expression, that includes the ways that creature uses its body to communicate, learn and survive.

Believability Through Aspects Besides Behaviour

With everything said so far, one assumption comes into place: a character's believability is mainly dependent on that character's behaviour. However, in [Camilleri et al., 2016], this core assumption is thrown out of the way, in order to study the impact of other variables in an agent's believability. In the work, the study focuses on how the game environment and other aspects of game design (such as level design) can affect the believability of a character. The game used as testbed was a variant of Infinite Mario Bros. The feature extraction process included both level design and gameplay features. At level design, the four features extracted were number of gaps in the level, average width of the gaps, number and placement of enemies in the level. Regarding gameplay features, a total of fourteen were extracted, and included variables like number of deaths, completion times, and number of jumps. Then, opinions of observers were obtained through online questionnaires about four different players playing the generated levels. Two of the players were controlled humans, and the other two were AI agents (that were created for the Mario AI Championship described previously, and reused in this work).

Then, with the use of RankSVM algorithm, the build of computational models that maps the features to the questionnaire responses was attempted. The model with best prediction results presents an accuracy of 73,31%, and is the model who takes into account both level design and gameplay features, which indicates a relationship between game environment and a player's believability perceived by observers. Therefore, the initial theory of game environment impacting an agent's believability is validated, at least, in the platformer genre.

2.3.4 Bottom-Up Modeling Approach: Player's Imitation

There are two main ways to model a player's in-game actions with the aim of imitating his behaviour [Ortega et al., 2013]. The most common one is direct imitation, in which the play traces of a human player are considered. Play traces are the "vestiges" or "evidences" that a human player leaves while playing a game. For instance, in a platform game like Infinite Mario Bros, the play traces of a player is the trail that he follows to overcome the obstacles and complete the levels. Hence, through supervised learning methods, the play traces are used as training sets, having the features of the game state as input, and having the chosen action by the player in that state as output.

The indirect method of player imitation is through reinforcement learning, where the goal is to optimize a reward function that measures human-like behaviour of an agent. The indirect way possesses one advantage over the direct one, that is the **generalization to unknown scenarios**. In fact, while training the agent using supervised learning, the agent's performance is based on whether it can imitate the player or not. In-game performance per si is not taken into account. Therefore, once the agent comes upon a scenario that is not in the training set, it will not know how to react. This leads to the agent presenting worse performance than the behaviour it was trained to mimic.

Generative Adversarial Imitation Learning

One particular imitation learning method is Generative Adversarial Imitation Learning (GAIL), that uses an adversarial approach in order to train the agent's policy to act similarly to the provided demonstrations. In this method, there is a second neural network called the discriminator, whose function is to distinguish whether an action is from a demonstration or performed by the training agent. The discriminator then acts as a "judge", and the agent's job is to "fool" it, i.e. the agent must mimic the demonstrations so perfectly that the discriminator thinks that the new observed state-action pair is from the demonstrations and not from the training agent. Therefore, the rewards provided by the discriminator are based on how much it believes that the observed state-action pairs are from the provided set of demonstrations.

GAIL works best when a limited number of demonstrations is available, and can also be used alongside with extrinsic rewards during the training process.

Related Works

Imitating human playing styles in Super Mario Bros One particular example of modeling a player's behaviour through imitation can be found in [Ortega et al., 2013]. The chosen representation of the environment was depicted with the use of two matrices, with dimensions 4x7, where one matrix contains information about the level and the other about the enemies nearby, at a precise moment of the game. A total of 65 environment variables were fed as input to an artificial neural network. Both imitation approaches were used, as well as a hand-coded rules-based agent. In total, six different controllers were implemented and tested. Three of the methods are based on backpropagation, neuroevolution and dynamic scripting, respectively. The hand-crafted one consists of a simple forward jumping agent, and the two remaining methods are taken from the "Mario AI championship", already talked about in the previous section 2.3.2.

The authors conclude that although none trained agent managed to resemble enough human-likeness to be able to trick the observers, agents trained to be human-like are closer to said goal (if their performance is good enough) than agents that simply are trained to win the game. More importantly, the authors conclude that the indirect way of player imitation achieved decisively better results than any other method.

HRLB[2] More recently, in [Cruz and Uresti, 2018], a hierarchical reinforcement learning framework is proposed to model believable bots for the video game *Street Fighter IV* (Capcom, 2009). With this framework, two open challenges in the creation of human-like NPCs are addressed: exploration of high-dimensional state spaces, and generation of behaviour diversity. The framework is based on MAXQ hierarchical decomposition, which consists of decomposing a Markov Decision Process (MDP) into a set of smaller semi-Markov Decision Processes (SMDP). A SMDP is a generalization of MDPS, including actions that take more than one timestep to execute. Therefore, this method allows to represent the problem as a task graph, in which the children of the root node represent playstyles, and their descendents represent subtasks. The leaf nodes denote primitive actions. With this representation, the problem of a high-dimensional state-action space is tackled. For the second problem, the framework adapts an off-line reinforcement learning that observes the player's actions in an encounter, and in the end of it, it updates the bot's model accordingly.

Results show that it was possible to create a human-like bot that can play the game at medium and advanced difficulty levels.

Deep Player Behaviour Modeling is proposed in [Pfau et al., 2020] for automated testing, to find in-game imbalances in encounters of the massive multiplayer online role-playing game *Aion* and how to fix them. The technique of deep player behaviour modeling consists in producing a replicative agent of an individual player through the mapping of the game states with the player's actions, just like a typical player behaviour imitation task. However, a feed-forward multilayer perceptron with backpropagation, that contains four hidden layers, is used for training. The activation function used was the logistic sigmoid. The dataset used was an agglomeration of the recording of gameplay sessions of 213 players during the course of 6 months. In *Aion*, several classes are available for the player to choose. Each player in the dataset corresponds to one of those in-game classes. To calculate imbalances between player classes, four variables were considered to obtain a proficiency metric: the result of the match (a binary value that either represents a win or a defeat), the duration of the match, and both the player's and the opponent's remaining health percentage.

The experiments involved putting all replicative agents against each other in a one-on-one fight, and against a collection of 100 opponents that incrementally increased in difficulty. The results obtained show the existence of discrepancies between classes proficiency, which suggests imbalance between them. In conclusion, deep player behaviour modeling shows great potential as an automated testing tool, being able to inform designers and developers of in-game imbalances.

Other works regarding player's behaviour imitation include [Gorman et al., 2006], and [Togelius et al., 2007]. In [Gorman et al., 2006], human behaviour is imitated through Bayesian methods for designing Quake (GT Interactive, 1996) bots. In[Togelius et al., 2007], neural networks are built to replicate player's behaviour like steering and thrust in a 2D car racing game.

2.4 Chapter Summary

In this chapter, several topics were approached.

First, an in-depth look of how AI can be linked with games was given. More specifically, four subtopics were explained.

The first subtopic was a general overview of how games can contribute to the AI research field. Games bring several benefits to the area of AI, namely understandability, public awareness, speed and relevant industrial application. Furthermore, games are hard and interesting problems, and some of the major AI milestones were achieved using games as case studies.

The second subtopic was about the wide range of games that AI can play. A game type commonly used as AI benchmarks is the arcade-style. The original video game Bomberman, which serves as inspiration for the case study of this dissertation, belongs to this category of games. However, since the case study is actually a turn-based adaptation of the arcade video game, it requires more strategy to be played than the original.

Next subtopic referred to the numerous AI techniques used to play games. An overview of the most common ones was given in this chapter, stating their pros and cons. Understanding how these methods work is crucial, allowing us to take a well-informed pick of techniques to apply in this dissertation's project.

The last subtopic explained the different roles that AI can take while playing games. The most common scenario in academia is using AI as a player to win the game, while playing as a NPC to improve the human player experience is the most used configuration in the game industry. The synthetic players developed for this dissertation present the role of a player in order to enhance human experience, which is mostly achieved by replacing human testers at evaluating new game scenarios.

The next subsection discussed simulation-based testing. Several previous works on this area of research are highlighted, like the continuous efforts of Holmgard et al. that demonstrate the usefulness of using agents as playtesters of generated content.

Synthetic players need to be believable, i.e, to behave like a human player would when facing the same situation. Synthetic players that do not follow this requirement may provide incorrect testing feedback. The two main approaches to model believable agents are top-down, borrowing frameworks from social sciences, and bottom-up, that consists of player's behavior imitation.

The chapter culminates with a brief overview of other related works using Bomberman as a case study. Some of them can be used as guidelines for the modeling of synthetic players, like the environment configuration and the neural network hyperparameters of [França et al., 2019] for a machine learning agent.

Chapter 3

Modeling Of Synthetic Players

This chapter explains how the two approaches were selected and implemented to develop synthetic players. The entire source code can be consulted at:

`https://github.com/Drca1997/Tese/tree/Diogo`

3.1 Selection of Approaches

Having the background provided by section 2.1.3 about possible AI techniques to play the game, a well-informed decision about which methods to choose can be made.

Ad-hoc methods like finite state machines and behaviour trees were discarded right away, due to their likelihood of producing a pattern of predictable behaviours, rendering non-human-like synthetic player, failing the third requirement previously listed. They also require fully hard-coded behaviour. Regarding finite-state machines, from an engineering standpoint, adding new actions and states to the game can also be cumbersome to implement.

Utility-based AI could be a viable option, since it can handle unique situations in a graceful manner and constantly weighs all the actions. Furthermore, utility-based systems possess a major advantage over finite-state machines and behaviour trees, that is the allowance of variations in behaviour. However, an utility-based approach was not ultimately chosen to model the synthetic players. because at the beginning of the design proposal, a deeper and more complex adaptation of the original *Bomberman* video game was intended to serve as the case study. Due to the multitude of proposed innovations in that adaptation, choosing values to qualify the overall desirability of a state or action would have been a harder task. Therefore, this approach was not selected. Looking retrospectively at the final output of the dissertation and the case study which the synthetic players were developed for, an utility-based system could have been, in fact, one of the implemented approaches.

The first chosen approach was a planning one. More precisely, A* algorithm to find the shortest path of a graph where nodes are game states and edges are possible actions for the synthetic player to execute. Planning was chosen for its good generalization capability. Due to the way the approach works (already described in section 2.1.3), new and unique situations can be well-handled, given that the action space remains the same. In fact, if there are no new available actions in the new scenario that the planning agent comes across, a solution will be found (if there is one available) even if the goal differs from the one of previously seen game scenarios. Implementing new actions is also a relatively easy task to do, because of the nature of an action's representation, that is dictated by preconditions and effects of the action in the game world.

One hindrance to take into account is the fact that replanning can become a computationally expensive task. Also, designing goals for the agent to fulfill the requirement of playing in a believable way may become a hard task to complete, due to the subjective nature of said requirement. A more in-depth look of how this approach works and was implemented is given in section 3.4.

The second chosen approach was machine learning, more specifically, RL. This method was chosen for its interesting ability of making the agent "learn" to play the game, therefore not having the necessity to hard-code behaviours. Machine learning also has the advantage of being relatively easy to set up, therefore diminishing the amount of coding required. To speed up the training process, first Imitation Learning (IL) is applied. Gameplay demonstrations of a human player are fed to the agent, which tries to mimic the human behaviours seen in the demos. Then, the performance of the agent is improved through deep RL.

IL provides a good starting point, since it reduces the search space but also allows to model the agent to behave in a "human-like" way, complying with the third requirement previously stated. Regarding the second requirement, a good generalization capability can also be achieved with this approach, since the training is done in procedural generated game scenarios. This will grant a greater degree of robustness to the agent.

Cons of this approach are the complete loss of designer control as well as being nearly impossible to edit and tune agent behaviours as desired. In addition, if the game state representation is misinterpreted, it can have serious repercussions on the agent's learning process. Details of how this approach works and how the training process was done are given in section 3.5.

3.2 Case Study

The case study that was used as a benchmark for this work is a turn-based adaptation of the *Bomberman* video game. *Bomberman* (Hudson Soft, 1985) is an arcade video game developed for *Nintendo Entertainment System* (NES), an 8-bit home video game console. In this game, the player needs to strategically place bombs on the map to detonate both enemies and blocks blocking the player's path. When detonated, a bomb creates an explosion that spreads in a cross-shape manner (vertically and horizontally) to neighbouring tiles (as long as the tile is not obstructed), occupying them with fire. If the fire reaches a tile that contains the player, an enemy or a destructible block, that entity is destroyed. There is a chance that a power-up appears on the tile if the destroyed game element was a block. These power-ups, when picked up by the player, grant special abilities like bombs with a larger explosion range. The possible actions of the player in this game are limited to move (up, down, left, right) and to plant a bomb in the current tile where the player stands on. The original game spurred the creation of several more versions of it, and in most of them there is a multiplayer mode. In this mode, the goal is to be the last player standing, by killing all the opponents.

In this adaptation, the main game rules still apply. More specifically:

- Players can place bombs on the map that will explode after a certain amount of timesteps.

- When detonated, bombs may kill players and destroy destructible blocks that are in the blast radius.
- Ultimate goal is to be the last player alive.

However, since the transition to a turn-based system will be made, several modifications must be in place to achieve this new style of play:

- First, players will not take actions at the same time, but in a predetermined order. This order is selected at the start of every new game turn and is random. (A new game turn starts after every player has taken an action).
- Each player may take **one action** on its turn. Possible actions are: **Move** to another tile, **place a bomb** and **do nothing**.
- Players can only move **one tile** per move action.
- A bomb explodes after **three timesteps** have elapsed.

Finally, for the sake of simplicity, power-ups were not incorporated in the case study.

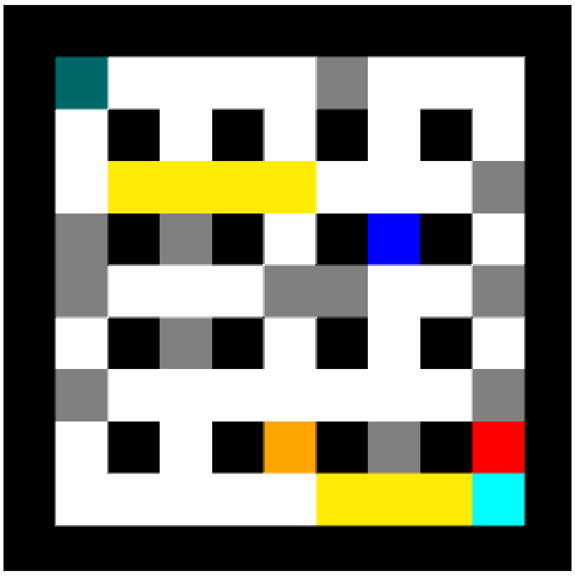

Figure 3.1: A match from the case study. Black blocks are unsurpassable. Grey blocks are destructible blocks. Red tiles are bombs. Yellow tiles are Flames caused by a Bomb explosion. Dark and light blue, green and orange blocks represent four different players.

A game scenario progresses as follows:

- First, the setup, where the grid is created and the agents are placed in the game world. The types of synthetic players that will play the game scenario are picked in this phase. This happens in the *Setup.cs* file, in the function *Grid SetupGrid(System.Random prng)*. Deeper details of the setup procedure are out of the scope of this dissertation, and can be found in [Correia, 2021].
- From the setup, a list of game agents is obtained. Game agents include any agent within the game, not only synthetic players. For instance: bombs, destructible blocks, fire (from bombs explosions) are all considered game agents, for the purposes of the work of [Correia, 2021].
- Each timestep, each agent in the list is updated. The order in which agents are updated is randomized each timestep.
- An agent being updated receives a grid that represents the current state of the game

world and evokes its *UpdateAgent()* method, executing whatever lays inside this function;
- In the case of the synthetic players, the *UpdateAgent()* method is responsible for asking the players for a decision of which action to take, initiating their decision making process.
- A synthetic player's action is then executed, modifying the received grid of the game world accordingly;
- Regarding human players, the game loop awaits for the player's input before proceeding to the next agent to be updated;
- After every game, the agent is updated, the game visuals are also updated to reflect the changes in the game world during the current timestep.
- This loop continues until the game is over. This may happen either if there is a winner or if the game reaches the predefined maximum amount of turns.

3.2.1 Bomberman-Related Works

This subsection lists some previous works that also used some version of *Bomberman* as case study.

Learning How to Play Bomberman with Deep Reinforcement and Imitation Learning

In [França et al., 2019], imitation learning and deep reinforcement learning are used to teach a bomberman agent to play the game. The authors use the Unity ML-Agents framework, and they test four different algorithms and five state representations.
The four tested algorithms were:

- Proximal Policy Optimization (PPO) integrated with a MLP;
- PPO integrated with LSTM neural network;
- Behavioral Cloning (BC);
- BC and then training with PPO;

The five tested state representations were the following:

- **Binary Flag**: for each cell of the grid, there is a bit flag that represents the content of the cell.
- **Normalized Binary Flag**: Similar to Binary Flag, but the observation vector values are normalized in order to be in $[0.0, 1.0]$ range.
- **Hybrid**: combination of the binary flag representation with One-Hot Vector representation(the latter is explained in detail in 3.5.1). This hybrid approach consists of each cell of the grid being represented by an observation vector of size equal to the total number of game elements that can be on a cell. For instance, if only agents and bombs are considered as game elements, the observation vector of each cell would have size of 2, in which each element would be binary, indicating the presence or absence of the game element associated with the respective vector index.
- **ICAART**: in this representation, each cell of the grid is characterized by a vector of four elements. The first element indicates what the tile contains, the second depicts the presence or absence of the player on that specific tile, and the third tile, similarly, states the presence or absence of an enemy on that cell. The last element of a cell's observation vector is used to represent the danger of the tile, according to the number of timesteps left in order for nearby bombs to explode.

- **ZeroOrOne**: similar to Hybrid representation, but each observation is added to the observation vector separately.

A tournament composed of 10 000 matches evaluated the performance of agents with each of the representations listed above. Then, the best representation was selected in order to train agents with the four types of algorithms. These agents were then pitted against each other in another 10 000 matches tournament Results show that usingBC before applying PPO can influence the training, and the Long-Short Term Memory (LSTM)-PPO coupling approach produced the best agents. Regarding state representations, the hybrid representation was the one achieving the best results.

BAIP

BAIP (Bomberman as an Artificial Intelligence Platform) [Lopes, 2016] is a language agnostic Bomberman-themed platform for educational purposes. The work also includes development of several agents for the platform, namely:

- Basic agents with simple heuristics;
- Search-based agent; more precisely an action planning agent;
- Reinforcement learning agent;

A conclusion that was drawn with the implementation of the search-based agent, is that the best maximum plan size to achieve the best performance is 2. The reinforcement learning agent presented modest results.

3.3 Synthetic Players General Architecture

Both types of synthetic players developed throughout this work inherit from the same abstract class, *SyntheticBomberman.cs.* This class contains functionalities that are common to both synthetic players. More specifically, it incorporates a method to process actions taken by the synthetic players, updating the game world accordingly to said actions. It also contains a method to convert the game world into a representation common for both agents. The representation itself is a grid of integers, in which each of them represents a possible map tile configuration. The following list presents an enumeration of every possible tile configuration:

- *Player*: Tile currently occupied by the synthetic player.
- *AIEnemy*: Tile currently occupied by a bomberman enemy.
- *Walkable*: Empty tile.
- *Destructible*: Tile that contains a destructible block.
- *Unsurpassable*: Tile that contains a block that cannot be exploded.
- *Bomb*: Tile that contains a bomb.
- *Fire*: Tile that was inside the blast radius of a nearby bomb, and therefore currently contains fire.
- *PlayerNBomb*: Tile that contains both the synthetic player as well as a bomb. This configuration only happens when the player plants a bomb and if the player stays in the tile.
- *AIEnemyNBomb*: Similar to *PlayerNBomb*, but for an enemy bomberman.
- *FireNExplodable*: Rare configuration that happens only in the timestep when an destructible block is exploded by a bomb explosion. In this timestep, the tile contains both the destructible block and fire.

- *FireNPlayer*: Tile that currently contains the synthetic player and fire because of the blast radius of a nearby bomb. If this configuration lasts until the end of a turn, the synthetic player dies. In other words, the bomberman dies if it ends its turn on a tile with fire.
- *FireNAIEnemy*: Similar to *FireNPlayer*, but for an enemy bomberman.
- *FireNBomb*: Tile that contains a bomb and fire. This configuration happens not only after a bomb detonates, but also if another bomb is inside the blast radius of the first bomb.
- *FireNBombNPlayer*: Rare configuration that happens if in a tile there is fire, a bomb, and the synthetic player. This only happens if the synthetic player plants a bomb on a tile inside the blast radius of another bomb.
- *FireNBombNAIEnemy*: Similar to *FireNBombNPlayer*, but instead of the synthetic player, it is a bomberman enemy.

While this representation is the one that the planning synthetic player uses for its decision making process, the machine learning agent does not use it as the final one, but as an intermediate representation. The grid is then transformed into a representation more suitable and efficient for the machine learning process, when the agent is collecting observations from the environment. This representation is explained in more detail in section 3.5.1 .

Figure 3.2 [1] shows the inheritances in place.

3.4 A Planning Approach to Synthetic Players

The planning agent is implemented in the *PlanningSyntheticPlayer.cs* file, and inherits from the *SyntheticBomberman* class previously described. This agent works by finding a sequence of actions that, from the current game state, will allow it to achieve a certain, selected goal.

3.4.1 Actions

An action is represented symbolically, with associated preconditions and effects. The preconditions are circumstances that need to be true, in order for the action to be possible, to be executed in the current game state. For instance, to plant a bomb, there cannot be any more bombs in the same tile. Effects are a grid representation (described previously in 3.3) of the game world after the action is performed. Actions also have an associated cost, although all of them possess the same value for the case study in use.

Every action inherits from the abstract class *SymbolicAction*, overriding five functions: *void Init(PlanningSyntheticPlayer agent)*, *void Simulate()*, *void Revert()*, *bool CheckPreconditions(int[,] grid)*, and *bool IsPossible(int[,] grid)*;

void Init(PlanningSyntheticPlayer agent) defines the Effects of the action.

void Simulate() virtually simulates the action upon the given state of the game, without actually executing it. This is useful in the A* search, where expanding a game world node means trying every possible action in that node.

[1]For a clear "waters separation": the *SyntheticBomberman* class inherits from a *GameAgentPlayer* class that was developed by Bernardo Correia, for his work [Correia, 2021]. The constructor of the *SyntheticBomberman* class was also developed by B. Correia. The *GameAgentPlayer* class also inherits from another one, *GameAgent*, also developed by B.Correia.

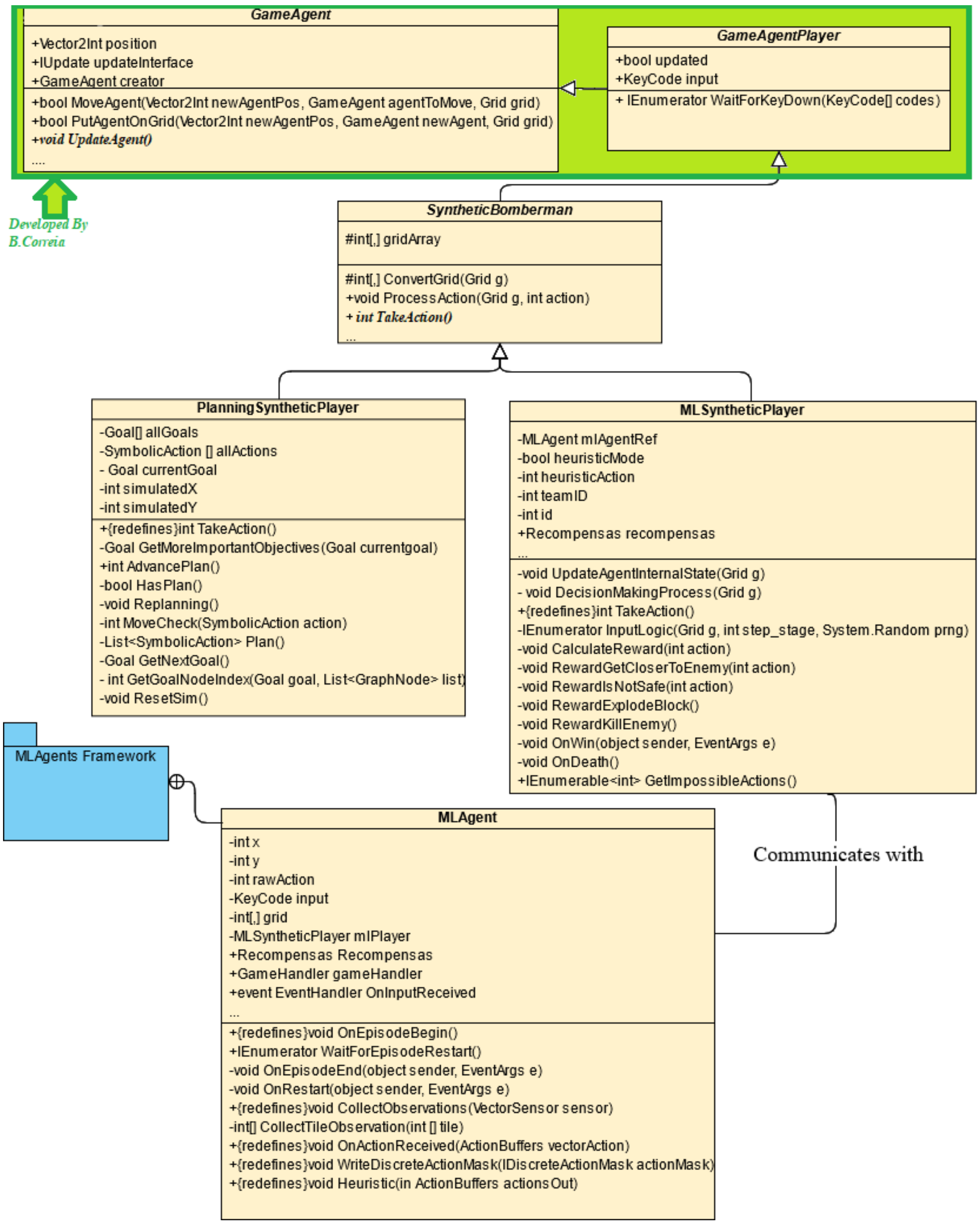
GameAgent
+Vector2Int position
+IUpdate updateInterface
+GameAgent creator
+bool MoveAgent(Vector2Int newAgentPos, GameAgent agentToMove, Grid grid)
+bool PutAgentOnGrid(Vector2Int newAgentPos, GameAgent newAgent, Grid grid)
+void UpdateAgent()
....
GameAgentPlayer
+bool updated
+KeyCode input
+ IEnumerator WaitForKeyDown(KeyCode[] codes)
Developed By B.Correia
SyntheticBomberman
#int[,] gridArray
#int[,] ConvertGrid(Grid g)
+void ProcessAction(Grid g, int action)
+ int TakeAction()
...
PlanningSyntheticPlayer
-Goal[] allGoals
-SymbolicAction [] allActions
- Goal currentGoal
-int simulatedX
-int simulatedY
+{redefines}int TakeAction()
-Goal GetMoreImportantObjectives(Goal currentgoal)
+int AdvancePlan()
-bool HasPlan()
-void Replanning()
-int MoveCheck(SymbolicAction action)
-List<SymbolicAction> Plan()
-Goal GetNextGoal()
- int GetGoalNodeIndex(Goal goal, List<GraphNode> list)
-void ResetSim()
MLSyntheticPlayer
-MLAgent mlAgentRef
-bool heuristicMode
-int heuristicAction
-int teamID
-int id
+Recompensas recompensas
...
-void UpdateAgentInternalState(Grid g)
- void DecisionMakingProcess(Grid g)
+{redefines}int TakeAction()
-IEnumerator InputLogic(Grid g, int step_stage, System.Random prng)
-void CalculateReward(int action)
-void RewardGetCloserToEnemy(int action)
-void RewardIsNotSafe(int action)
-void RewardExplodeBlock()
-void RewardKillEnemy()
-void OnWin(object sender, EventArgs e)
-void OnDeath()
+IEnumerable<int> GetImpossibleActions()
MLAgents Framework
Communicates with
MLAgent
-int x
-int y
-int rawAction
-KeyCode input
-int[,] grid
-MLSyntheticPlayer mlPlayer
+Recompensas Recompensas
+GameHandler gameHandler
+event EventHandler OnInputReceived
...
+{redefines}void OnEpisodeBegin()
+IEnumerator WaitForEpisodeRestart()
-void OnEpisodeEnd(object sender, EventArgs e)
-void OnRestart(object sender, EventArgs e)
+{redefines}void CollectObservations(VectorSensor sensor)
-int[] CollectTileObservation(int [] tile)
+{redefines}void OnActionReceived(ActionBuffers vectorAction)
+{redefines}void WriteDiscreteActionMask(IDiscreteActionMask actionMask)
+{redefines}void Heuristic(in ActionBuffers actionsOut)

Figure 3.2: UML Class Diagram

void Revert() simply reverts what *void Simulate()* did, so one can easily return to the parent node in the A* search, after expanding its children.

bool CheckPreconditions(int[,] grid) is called during A* algorithm execution to check if an action is possible to execute in the current game state.

bool IsPossible(int[,] grid) has the same function as *bool CheckPreconditions(int[,] grid)*, however it is used when the planning synthetic player is checking if the next action of the current plan is possible to perform.

The full set of actions that exist are:

- *ActionMoveUp*: moves the agent up one tile;
- *ActionMoveDown*: moves the agent down one tile;
- *ActionMoveLeft*: moves the agent left one tile;
- *ActionMoveRight*: moves the agent right one tile;
- *ActionPlantBomb*: plants a bomb on the tile where agent is currently in;
- *ActionDoNothing*: does nothing, skipping current turn;

3.4.2 Goals

A goal is a desired game state. Goals inherit from the same abstract class *Goal*, overriding four functions: *bool IsPossible()*, *double Heuristic(WorldNode state, WorldNode goal)*, *bool IsObjective(WorldNode node)*, and *int[,] GetGoalGrid(int[,] currentGrid, int index, PlanningSyntheticPlayer agent).*

bool IsPossible() checks if the goal is still possible to achieve given the current configuration of the game world. This function is called when the planning synthetic player is checking if the goal is still possible before executing the next action of the current plan.

double Heuristic(WorldNode state, WorldNode goal) rovides a custom heuristic function to use in the A* algorithm.

bool IsObjective(WorldNode node) checks if the given game world node is the goal node. This function is called during the execution of the A* algorithm.

int[,] GetGoalGrid(int[,] currentGrid, int index, PlanningSyntheticPlayer agent) builds and returns the grid representation of the goal. This function is called by the planning synthetic player that then passes the obtained grid to the A* algorithm as an input, representing the start node of the search (since A* applies regressive search, as explained in detail in section 3.4.4).

There are three existing goals in total:

- *BeSafeGoal*: goal of the highest priority, in which the agent checks if it is currently in a safe tile, i.e, in a tile away from the explosion radius of every bomb currently placed in the game world;

- *AttackEnemyGoal*: goal of attacking an enemy, that consists in approaching the targeted enemy and planting a bomb near it.

- *ExplodeBlockGoal*: goal of the lowest priority. If the agent is already safe and it cannot find a path to any enemy, it will decide to explode the nearest block, in hopes of finding a passage to enemies;

3.4.3 Planning Synthetic Player's Model

Attributes

An instance of the planning synthetic player contains the following attributes:

- *private Goal[] allGoals*: an array with all the goals the agent can eventually pursue during a game session

- *private SymbolicAction [] allActions*: an array with all the actions the agent can eventually perform during a game session.
- *private List<SymbolicAction> currentPlan*: the current plan the agent is executing, which is a sequence of actions.
- *private Goal currentGoal*: the current goal the agent is pursuing.
- *private int simulatedX*: X coordinate of the simulated position of the agent.
- *private int simulatedY*: Y coordinate of the simulated position of the agent.

The simulated position of the agent is a fictitious pair of coordinates that states where the agent is on the grid while the planning process is being executed. These are the coordinates that are affected by an action's effects. With the existence of this simulated position, twinkling with the agent's real position during planning is avoided. Therefore, the agent's real position is only altered after executing the next action of the generated plan, and the simulated position reverts back to being equal to the real position. The simulated position is then only used in calculations again when a new plan needs to be generated.

Constructor

The agent's construtor presents the following parameters:

- *List<int> states*: a list of internal states that every game element possesses. The planning agent does not use this, since it is intrinsic to the platform's inner workings.
- *int x*: the x coordinate of the agent's position in the grid;
- *int y*: the y coordinate of the agent's position in the grid;
- *IUpdate updateInterface*: reference to the platform's update interface, responsible for the game loop update.
- *Goal [] allGoals*: a vector of all the possible goals the agent can pursue during a game session;
- *SymbolicAction [] allActions*: a vector of all the possible actions the agent can perform during a game session;

Decision Making Process

When asked for a decision, the agent either returns the next action of its current plan, if there is one, or replans and returns the first action of the new plan. The agent checks the following conditions:

- Is there currently a plan?
- Is not there any higher-priority goal possible to pursue than the current one?
- Is the current goal still possible to achieve?

If all the answers to these conditions are affirmative, then the agent checks if the next action of the current plan is possible and if it can be performed safely, i.e, without the risk of the action causing the demise of the agent. If yes, then the action is executed. If not, the agent decides to do nothing on the current turn.

If at least one of the previous conditions are not met, then the agent needs to replan. The replanning process goes the following way:

1. The agent checks for the possible goal with highest priority
2. A* for pathfinding in order to decide the goal tile. For some goals, several tiles can

be considered goal tiles. For instance, for the goal of attacking an enemy, any tile adjacent to the targeted enemy can be considered a goal tile for the agent, i.e., a tile where the agent needs to go in order to complete the goal. However, regressive search is done when planning, which means it starts on the goal node and finds a path to the current node. Therefore, there is the necessity of defining a specific tile, from the several possible options, to be considered as the starting node of the search. For sake of simplicity, pathfinding is done from the current tile to the tile where the enemy is, in order to get the nearest adjacent tile to the enemy. The obtained tile is then used as the starting node of A* planning search.

3. A* regressive search from goal node to current node. (explained in more detail in section 3.4.4).
4. The agent elaborates the plan and returns its first action, if it is safe to execute. If not, the agent decides to do nothing this turn and to initiate the newly generated plan only on the next turn.

Figure 3.3 shows the flow of the described process.

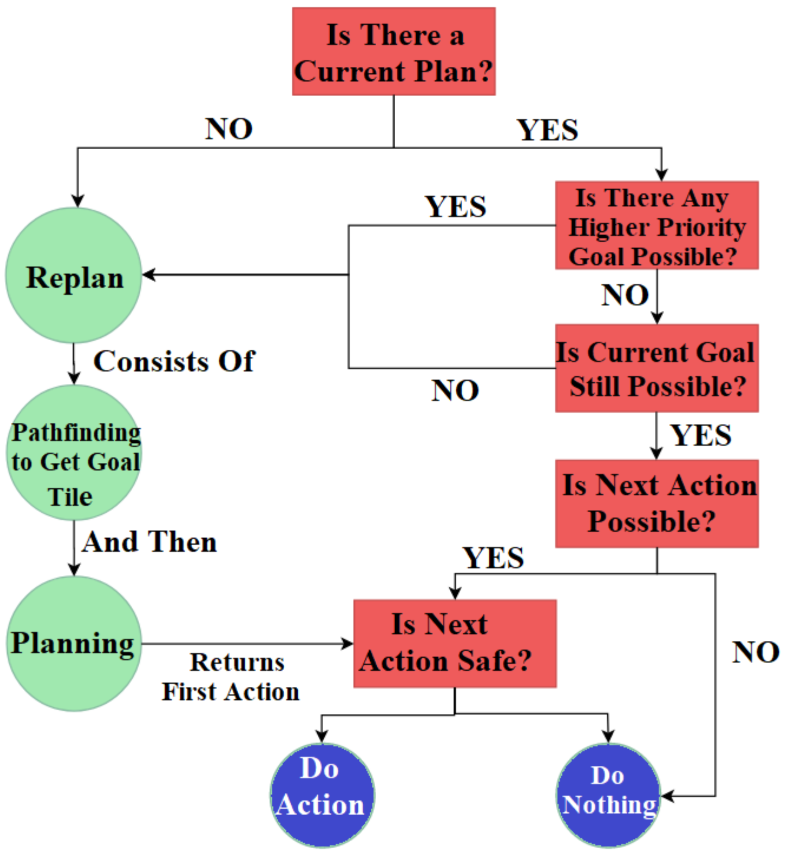

Figure 3.3: The Planning Synthetic Player's Decision Making Process

3.4.4 A* Search

The planning synthetic player uses the A* search algorithm to find a sequence of actions that fulfill the current goal - a **plan**. The search is performed in a regressive manner, in which the starting node of the search actually corresponds to the desired game state, and the goal node represents the current game state. This is to restrict the number of expanded nodes during the search process, which leads to a more efficient implementation of the algorithm.

Therefore, starting from the goal node, the algorithm checks for a possible action to execute that will lead to a new state, closer to the goal node of the search, which is the current game state. As said in 3.4.1, in order for an action to be possible to execute, all its preconditions need to be true.

A search node represents a game state, and is an instance of the WorldNode class, which inherits from GraphNode class. The latter represents a typical graph node used in A* search, that contains the following relevant attributes:

- *double gCost*: represents the $g(n)$ of the node, that is the cost from the starting node until the current node;
- *double hCost*: represents the $h(n)$ of the node, i.e. the estimated distance from the current node to the goal node;
- *double fCost*: represents the $f(n)$ of the node, that is calculated by the sum of $g(n)$ with $h(n)$;
- *GraphNode previousPathNode*: reference to another node. This is used after the execution of the algorithm, to trace back from the goal node to the start node, in order for the algorithm to be able to return the full path of nodes;

An instance of *WorldNode* is just an extension of *GraphNode*, that, alongside the attributes already detailed, it also contains:

- *int[,] grid*: a bidimensional matrix of integers that represents the game state that the node itself denotes;
- *int [] agentsPos*: a vector of two integers that denotes the coordinates of the agent in the game state represented by this node;
- *SymbolicAction actionTakenToReachHere*: reference to the Action taken to reach this goal. This attribute has the same function of the *previousPathNode* attribute of the *GraphNode* class;

The algorithm follows the typical A* execution, in which the search is done by visiting and expanding the nodes with the lowest $f(n)$ cost. It receives the following parameters as inputs:

- *PlanningSyntheticPlayer player*: reference to the planning agent that is evoking the A* algorithm;
- *GraphNode start*: the starting node of the search, that represents the goal game state
- *GraphNode end*: the goal node of the search, that represents the current game state;
- *SymbolicAction[] possibleActions*: array of all the possible actions the agent can perform;
- *Goal goal*: the current goal the agent is pursuing;

First, the starting node's $h(n)$ and $f(n)$ are calculated and then it is added to the list of possible nodes to expand (in [pseudocode], called "Expandable"). Then, while there are nodes to be expanded, the node with lowest $f(n)$ is selected and the *SimulatedX* and *SimulatedY* coordinate attributes of the planning agent are updated, with the virtual agent's coordinates on the selected node being assigned to them. For example: in the first iteration of this process, the selected node is the start node, since it is the only node on the expandable list. Therefore, the agent's simulated position will be updated to reflect its hypothetical position if the agent was in the game state corresponding to that node. Since the start node represents the actual goal game state, the agent's simulated position will be its position if the agent was in the game state corresponding to the goal node. These simulated coordinates ensure that there is no need to alter the agent's real coordinates during planning, composing a very basic version of a forward model, i.e, a way of simulating the effects of an action in the environment, resulting in a new game state, without actually modifying the real game world.

Next, the algorithm checks if the selected node is the goal node. If it is, then the *List<SymbolicAction> GetPlan(WorldNode node)* method is invoked, which traces back from the selected node to the start node, in order to get the full path. This traceback is possible due to the *actionTakenToReachHere* attribute that every node possesses. After the traceback is done, the *List<SymbolicAction> GetOppositeActions(List<SymbolicAction> actions)* method is invoked, still within *GetPlan(WorldNode node)*. This *GetOppositeActions(List<SymbolicAction> actions)* method obtains the reverse actions of the full action sequence from the trace back. This is necessary since the algorithm performs regressive search, which translates into a sequence from the goal game state to the current game state. So, for instance, let us consider the scenario depicted in Figure 3.4.

A planning agent (blue square) is at coordinates $(8, 1)$, and an adjacent bomb (red square) at $(7, 1)$. The agent's goal is to evade the bomb explosion, therefore it finds a path from the nearest safe tile to its current position at $(8, 01)$. Since the bomb explosion will reach tiles $(8, 1)$ and $(9, 1)$, and there are obstructing blocks (black squares) in $(8, 0)$, $(8, 2)$ and $(10, 1)$, the nearest safe tile is $(9, 2)$. In this situation, the A*'s regressive search will obtain the following plan: *Down-Left*, since it starts from the goal node on $(9, 2)$, to the current game state, in which the agent is at $(8, 1)$. Therefore, there is the need to not only reverse the entire plan's order, but also to get its opposite actions. So, first, the plan is reversed, thus obtaining *Left-Down*. Then, in order to get the opposite action for each one of the sequence, the attribute *SymbolicAction OppositeAction* of each action is used. This attribute is defined *à priori* and is a reference to another action. For instance, the opposite action of going left is going right, while the opposite move of going down is going up. Therefore, executing *List<SymbolicAction> GetOppositeActions (List<SymbolicAction> actions)* upon the sequence of actions *Left-Down* will render *Right-Up*, which is the correct sequence in order for the agent to go from its current state at $(8, 1)$ to $(9, 2)$.

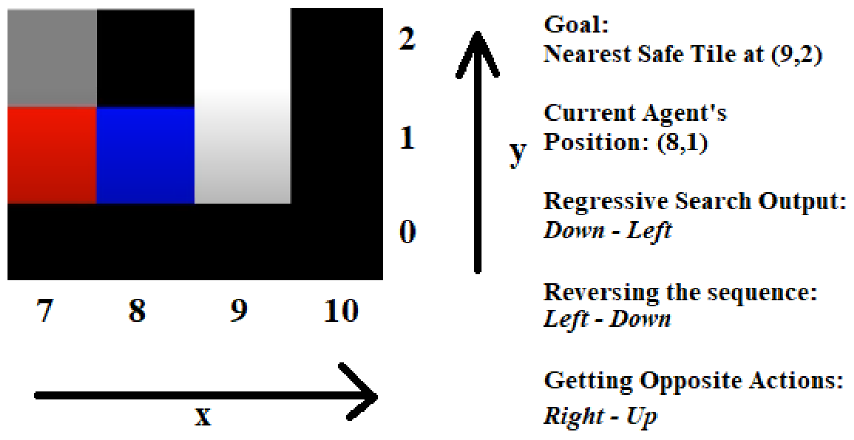

Figure 3.4: An Example of How Regressive Search Works

The program's execution ends returning the full sequence of actions in order to go from the current game state to the goal game state.

However, if the node that is selected from the expandable list is not the goal node, then it is removed from the expandable list and added to the one of visited nodes, being expanded in the process. The expansion is made by obtaining every possible action possible to be executed by the synthetic player on the game state depicted by the expanding node. Then, for each possible action, its effects are simulated upon the node's game state. This is done invoking the *void Simulate()* method intrinsic to each action. For instance, in order to simulate the action *Move Right*, the agent's *SimulatedX* attribute is updated accordingly, adding +1 to it. The previous tile depicted by the agent's simulated coordinates is updated as well in order to reflect the agent's hypothetical movement away from that tile. A new node is created following the simulation of the action's effects. The new obtained node is then checked to find out if it is present in the visited list or not. If it is, the action's effects are reverted in order to get the parent node's grid back and the rest of the process is ignored, passing automatically to the next action to be simulated. If the new node is not in the visited list, its $g(n)$ is calculated and its $f(n)$ is updated accordingly. Its $h(n)$ is also calculated if it has not been yet. Next, the new node is checked to find out if it is present in the expandable list or not. If it is, then only if the new $f(n)$ is lower than the current one, both $f(n)$ and $g(n)$ get updated. The *actionTakenToReachHere* attribute is assigned with a reference of the action that was simulated. If however, the new node is not present in the expandable list, then it is added to it, with both its $f(n)$, $g(n)$ and *actionTakenToReachHere* attribute being updated as well. After this, the action's effects are reverted in order to get the parent node's grid back, and then the next possible action to execute is simulated as well, repeating the entire process.

Pseudocode for the algorithm is presented in the Algorithm Listing 1, to better illustrate all of the process described above.

3.5 A Reinforcement Learning Approach to Synthetic Players

The RL approach was implemented recurring to Unity ML-Agents framework [2] [3].
The core idea of this approach is to perform IL and then improve the result with classic RL. More precisely, the first step is to provide the agent with gameplay demonstrations of a human player in order to train an agent's policy to mimic the behaviours seen in the demos. After the IL process concludes, the agent's performance is improved by doing more training, this time using only the extrinsic reward signals from the environment.

To perform IL, two methods were used: the first is BC, the simplest form of imitation learning, in which the goal is to train a policy that maps state-action pairs from the provided demonstrations. The second used method is GAIL, which was explained in 2.3.4. To perform RL, the framework provides two algorithms to choose from. The algorithms in question are PPO[4] and Soft Actor Critic (SAC)[5].

PPO is the default algorithm of the framework, since it is more general and more stable than other RL algorithms. PPO is a stochastic gradient descent algorithm and its novelty resides in the fact that it implements a way to do a Trust Region update without having the need to use the *Kullback-Leibler* penalty[6]. A Trust Region update is a way of making sure that the new policy does not deviate too much from the new one, in order to avoid learning completely off-the-book behaviours. This elimination of the need to use the KL-divergence allows PPO to achieve the same performance of other RL algorithms despite

[2]ML-Agents Framework Official Page: https://unity.com/products/machine-learning-agents
[3]ML-Agents Framework GitHub: https://github.com/Unity-Technologies/ml-agents
[4]OpenAI's Official Page about PPO: https://openai.com/blog/openai-baselines-ppo/
[5]SAC's Offical Announcement Page: https://bair.berkeley.edu/blog/2018/12/14/sac/
[6]Kullback-Leibler Penalty: https://en.wikipedia.org/wiki/Kullback%E2%80%93Leibler_divergence

Algorithm 1: A* For Planning

```
Input Planning synthetic player, Start node, End Node, All Actions, Current Goal
Expandable = Start Node;        // List of Nodes possible to be expanded
Visited = empty;                // List of already visited Nodes
Calculate $f(Start)$;           // $f(n) = g(n) + h(n)$
while there are nodes in Expandable do
    Node n = Get Lowest F-Cost Node in Expandable
    if Node n is Objective then
        return Get Plan from Node n to Start Node;
    end
    Remove Node n from Expandable and Add it to Visited
    Possible Actions List = Get Possible Actions to Execute in node N
    foreach Action in Possible Actions do
        Simulate Action
        Create New Node with the Effects of Simulated Action
        if New Node in Visited List then
            continue
        end
        Calculate $f(NewNode)$
        if New Node in Expandable List then
            if new $f(NewNode)$ lower than Old $f(NewNode)$ then
                Update $f(NewNode)$;
            end
        else
            Add New Node to Expandable List;
        end
        Revert Action to return to previous Node
    end
end
return failure
```

being far simpler to implement.

SAC is an off-policy algorithm, which means that the experiences collected during training are placed in a replay buffer and then they are randomly drawn from the buffer in future model updates. This allows the agent to learn from experiences collected at any time during the past. Despite being a sample-efficient algorithm, SAC is more computationally expensive than PPO, and is mostly used for real-world robotic problems. Therefore, PPO was chosen to perform RL, not only for its advantages but also because the framework documentation recommends using this particular algorithm when coupling RL withIL.

3.5.1 Representation

As said previously, the machine learning agent does not use the grid representation described in 3.3 as the final, but as an intermediate one. Instead, when collecting observations from the environment, the agent translates the grid representation to a modified version of the One-Hot Vector representation.

The One-Hot Vector representation is a vector of size equal to the total number of possible configurations, in which each element represents a possible configuration. The entire vector presents 0 in every index except for a single 1 on the index corresponding to the pretended configuration or state to be identified.

Using the One-Hot observation method for state representation would render a large amount of observations per time step, since there would be a one-hot observation vector for each cell of the world grid. Therefore, the total number of observations would be given by the total number of cells of the world grid multiplied by the number of possible tile configurations plus 2 (that represents the X and Y of the agent's position in the world). This is a rather computationally expensive approach. Therefore, a modification is needed to be made in order to diminish the number of observations. With the new representation, each tile is still represented by an observation vector. However, this vector has a size equal to the total number of types of game elements in the game. In the case study presented, there are five types of game elements:

- Bomberman (the synthetic players and respective enemies)
- Destructible block
- Unsurpassable block
- Bomb
- Fire

Thus, each grid cell is represented by a vector of size 5, in which each element represents the presence or absence of a particular type of game element. The indexes of the vector correspond to the following game element types:

$$[Bomberman, DestructibleBlock, UnsurpassableBlock, Bomb, Fire] \tag{3.1}$$

Therefore, for instance, a vector of $[1, 0, 0, 0, 1]$ represents a cell where there is a bomberman but also fire from a bomb explosion, which will eventually result in the death of the bomberman on the next timestep. A vector of $[0, 0, 0, 0, 0]$ represents an empty tile.

With this representation, the total number of observations is given by the total number of grid cells multiplied by the total number of game elements types +2 (agent's position). Since the majority of the training process happened on a 11x11 grid, and as stated above, there are 5 different types of game elements, the total number of observations is 11 x 11 x 5 + 2 (agent's position) = 607. The One-Shot representation, for the same map configuration, would be 11 x 11 x 15+2 = 1817. Therefore, the new devised representation makes the total number of observations three times smaller than the One-Shot observation representation.

3.5.2 RL Synthetic Player's Model

The synthetic player is represented by two scripts, *MLSyntheticPlayer.cs* and *MLAgent.cs*. The former inherits from the *SyntheticBomberman* class described in 3.3, while the latter inherits from a special class *Agent* specific to the ML-Agents framework.

The ML Synthetic Player Script

The *MLSyntheticPlayer.cs* file not only deals with the typical logic of a game agent of the platform, such as the construtor, the *UpdateAgent* function and the method to deal with the death of the agent, but also manages the rewards of the agent. It also deals with the Heuristic Mode, that is used instead of the default behaviour (training) of the synthetic player when one wants to provide gameplay demonstrations for the posterior imitation learning process. The *MLSyntheticPlayer* script manually asks, every turn, the *MLAgent* one for a decision to take.

A new instance of a machine learning synthetic player is created using the constructor shown below, in which the parameters are:

- *List<int> states*: similarly to the planning agent, this list of internal states is not actually used.
- *int x*: the x coordinate of the agent's position in the grid;
- *int y*: the y coordinate of the agent's position in the grid;
- *IUpdate updateInterface*: reference to the platform's update interface, responsible for the game loop update;
- *MLAgent agentRef*: reference to the *MLAgent.cs* script responsible for managing the machine learning process;

```
public MLSyntheticPlayer(List<int> states, int x, int y,
IUpdate updateInterface, MLAgent agentRef) :
base(states, x, y, updateInterface)
{
    ...
}
```

Listed below are the two main functions of this script. *void DecisionMakingProcess()* simply manages the natural reinforcement learning loop: it obtains the action to execute, processes the effects of the chosen action in the environment and collects rewards. To obtain the action, the int TakeAction() method is invoked, which reads the action to take from a variable called *rawAction* in the *MLAgent* script.

```
private void DecisionMakingProcess(Grid g){
    int actionTaken = TakeAction();
    ProcessAction(g, actionTaken);
    CalculateReward(actionTaken);
}

public override int TakeAction(){
    if (!heuristicMode)
        MlAgentRef.RequestDecision();
    return MlAgentRef.RawAction;
}
```

The ML Agent Script

The *MLAgent* script is where the machine learning process actually happens.
The first important method is *void CollectObservations(VectorSensor sensor)*, that gathers information about the environment at every timestep and builds the representation of the current game world.

```
public override void CollectObservations(VectorSensor sensor){
    base.CollectObservations(sensor);
    // Add Grid to Observation Vector
    foreach (int[] tile in SyntheticPlayerUtils.GridIterator(grid))
    {
        /*
        CollectTileObservation() returns a 5-sized
        vector that represents the tile in question
        */
        int [] tileState = CollectTileObservation(tile);
        for (int i = 0; i < tileState.Length; i++){
            sensor.AddObservation(tileState[i]);
        }
    }
    // Add Normalized X of Agent to Observation Vector.
    sensor.AddObservation(x / grid.GetLength(0));
    // Add Normalized Y of Agent to Observation Vector.
    sensor.AddObservation(y / grid.GetLength(1)); /
}
```

The obtained representation is the one explained in detail in section 3.5.1

The *void WriteDiscreteActionMask(IDiscreteActionMask actionMask)* function diminishes the amount of possible actions that the agent can try at every timestep, providing it with the list of impossible actions to execute in the current game state.

```
 public override
 void WriteDiscreteActionMask(IDiscreteActionMask actionMask){
    actionMask.WriteMask(0, mlPlayer.GetImpossibleActions());
}
```

After collecting observations and deciding which action to take, the agent calls the *void OnActionReceived(ActionBuffers vectorAction)* function, that stores the action to be taken and invokes an C# event. This event signals the *MLSyntheticPlayer* script that it can collect the action to execute in the current turn (by invoking *void DecisionMakingProcess()* and consequently, *int TakeAction()*) from the variable *RawAction* in the *MLAgent* script.

```
public override void OnActionReceived(ActionBuffers vectorAction)
{
    //stores the action to execute in rawAction
    rawAction = vectorAction.DiscreteActions[0];
    OnInputReceived?.Invoke(this, EventArgs.Empty);
}
```

The script also contains upkeep functions like *OnEpisodeBegin()* and *OnEpisodeEnd()*, that handle the logistics of starting a game scenario.

3.6 RL Synthetic Player's Training Process

This section describes the entire training process performed to train synthetic players. One training episode corresponds to a match of the case study. The match ends when there is a winner or when the maximum number of turns is reached.

3.6.1 Rewards

During the training process, several rewards were tested. The most used ones are presented in Table 3.4

Event	Reward
Winning the game	1.0
Dying	-1.0
Killing an enemy	0.75
Iteration penalty	-0.0001

Table 3.1: Main Rewards Used in The Training Process

In some training trials, there were also tested other rewards, as in [França et al., 2019], that can be consulted in Table 3.2

Event	Reward
Getting closer to the nearest enemy	0.002
Getting away from the nearest enemy	-0.002
Exploding a block	0.1
On a danger tile (within the blast radius of a nearby bomb)	-0.0001

Table 3.2: Other Rewards Used in The Training Process

3.6.2 Hyperparameters

The most important hyperparameters used in the training process are the following:

- *Batch_size*: Batch size is the number of experiences used in each iteration of the gradient descent algorithm, and it needs to be multiple times smaller than the buffer size.
- *Buffer_size*: Buffer size, in PPO algorithm, is the number of experiences to gather before updating the policy model. Must always be multiple times larger than the batch size.
- *Learning_rate*: Initial learning rate for gradient descent. The learning rate dictates the amount that the neural network's weights are updated during training.
- *beta*: β refers to the entropy regularization's strength. The higher its value is, the more probable it is for the agent to take random actions to explore the environment.
- *epsilon*: ϵ dictates the policy's evolution speed during training. In other words, it defines the acceptable interval for the divergence between the new policy update and the old one.
- *lambd*: λ defines how much the agent follows his current policy or the environment rewards when updating its value estimate.
- *num_epochs*: Number of epochs is the number of passes through the experience buffer when performing gradient descent optimization.
- *normalize*: Whether normalization is applied to the observation vector or not.
- *hidden_units*: number of units per hidden layer of the neural network.
- *num_layers*: number of hidden layers of the neural network.
- *gamma*: γ is the discount factor.

A typical configuration file presents a structure like the one depicted next. This configuration below can be assumed that was used in every trial, except when stated otherwise.

```
behaviors:
  MLBomberman:
    trainer_type: ppo
    hyperparameters:
      batch_size: 256
      buffer_size: 10240
      learning_rate: 3.0e-4
      beta: 5.0e-4
      epsilon: 0.2
      lambd: 0.95
      num_epoch: 3
      learning_rate_schedule: linear
    network_settings:
      normalize: false
      hidden_units: 128
      num_layers: 2
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
```

3.6.3 Machine Learning Trials

1st Series of Trials - Testing the Waters

The first machine learning training trials were done in order to "test the waters", i.e., to test the correct working of the framework and also the correct integration within the platform. Therefore, only two ML runs were executed, one dedicated to IL, another one to RL improvement over the former.

The first trial was focused on IL, more precisely on GAIL. The agent was put in an environment with 60% destructible blocks, and pitted against two synthetic players that simply execute a random action every turn, and an idle one, which returns the action *Do Nothing* every timestep.

The environment configuration can be seen in Tables 3.3 and 3.4.

Parameters	Values
% of Destructible Blocks	60%
Opponents	2 Random and 1 Idle
Provided Demo	20 Episodes in an 60% destructible blocks environment

Table 3.3: Environment Configuration of 1st Trial of 1st Series

The configuration of hyperparameters is the one depicted in 3.6.2, with the addition of the following lines under the *reward_signals* section, to activate GAIL.

```
    gail:
        strength: 0.5
        demo_path: Demos/[name of Demo].demo
```

Therefore, GAIL was used with a strength of 0.5.

The second trial had the objective of improving upon the first through classic reinforcement learning. For this purpose, the trial presented the exact same configuration as the first

Reward	Value
Winning the game	1.0
Dying	−1.0
Killing an enemy	0.75
Iteration penalty	−0.0001
Getting closer to the nearest enemy	0.002
Getting away from the nearest enemy	−0.002
Exploding a block	0.1
On a danger tile (within the blast radius of a nearby bomb)	−0.0001

Table 3.4: Rewards Used in The First Trial of 1st Series

one, except that GAIL strength was reduced to 0.1, in order to assign less power to the demonstrations, so the agent can learn more through the environment rewards.

The results of both trials are presented in Figure 3.5.

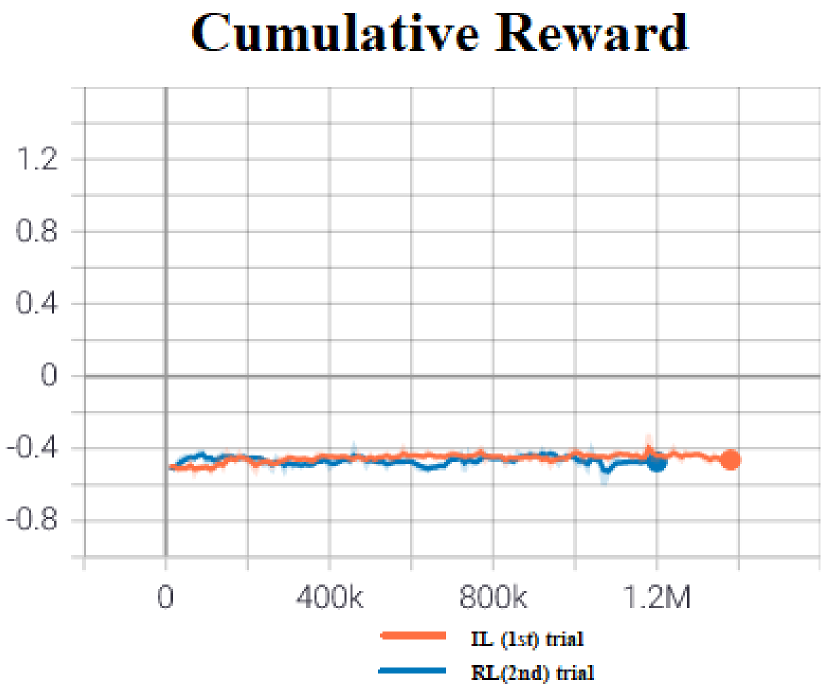

Figure 3.5: Results of the 1st Series of Trials

In the end of this series, despite not achieving relevant results, the main goal of the series was fulfilled: to test the ML-Agents framework and to perform some initial machine learning runs.

2nd Series of Trials - How The Environment Configuration Affects The Learning Process

The second series of trials had the main goal of studying how the performance of the agent would vary according to the percentage of destructible blocks in the environments. Intermediate rewards were also eliminated, leaving only the end game rewards and a reward

for killing an enemy.
In total, this series presents 9 trials, divided in 3 batches of 3 trials.

The first three trials consisted of imitation learning via BC plus GAIL, in three different environment configurations: 0%, 30% and 60% of destructible blocks, respectively. Three separate demos were recorded, consisting of 100 episodes played by a human player, one in a 0% destructible blocks map, another in 30%, and the other in 60%. Each trial got the demo with the corresponding percentage of destructible blocks to learn from. The agent faced three idle opponents during training.

The rewards used in the trials are the ones of 3.5.

Reward	Value
Winning the game	1.0
Dying	-1.0

Table 3.5: Rewards Used in The 1st Batch of Trials of 2nd Series

The imitation learning parameters used in the trials are the ones of 3.6.

IL Parameter	Value
BC Strength	1.0
GAIL Strength	0.5

Table 3.6: IL Parameters Used in The 1st Batch of Trials of 2nd Series

The introduction of BC to the configuration file is analogous to GAIL, except that is not put under *reward_signals* section. Therefore, the rewards part of the configuration file looks like the following:

```
reward_signals:
  extrinsic:
    gamma: 0.99
    strength: 1.0
  gail:
    strength: 0.5
    demo_path: Demos/[name of Demo].demo
behavioral_cloning:
  strength: 1.0
  demo_path: Demos/[name of Demo]].demo
```

The remaining hyperparameters were the same of 3.6.2.

Figure 3.6 shows the comparison between the three trials.

The second batch consisted of three trials was similar to the previous three, with the same exact setup, with the exception of the reintroduction of a reward for killing an enemy, with value 0.75. The results can be consulted in 3.7.

The last three trials consisted of improving the previous three (the second batch of trials of this series), lowering GAIL and BC strength to 0.1 in order to train the agent's policy based on the environment's extrinsic rewards. Rewards were the same ones used in the second batch of trials.

Results of the last batch of trials can be found in Figure 3.8.

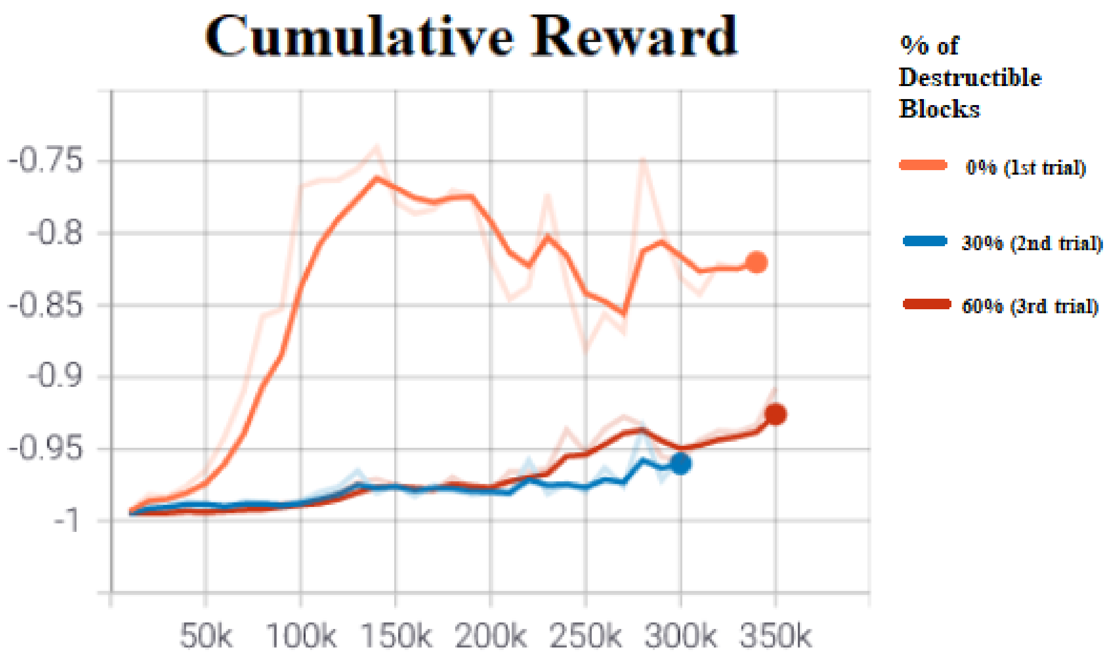

Figure 3.6: Results of The 1st Batch of Trials of 2nd Series

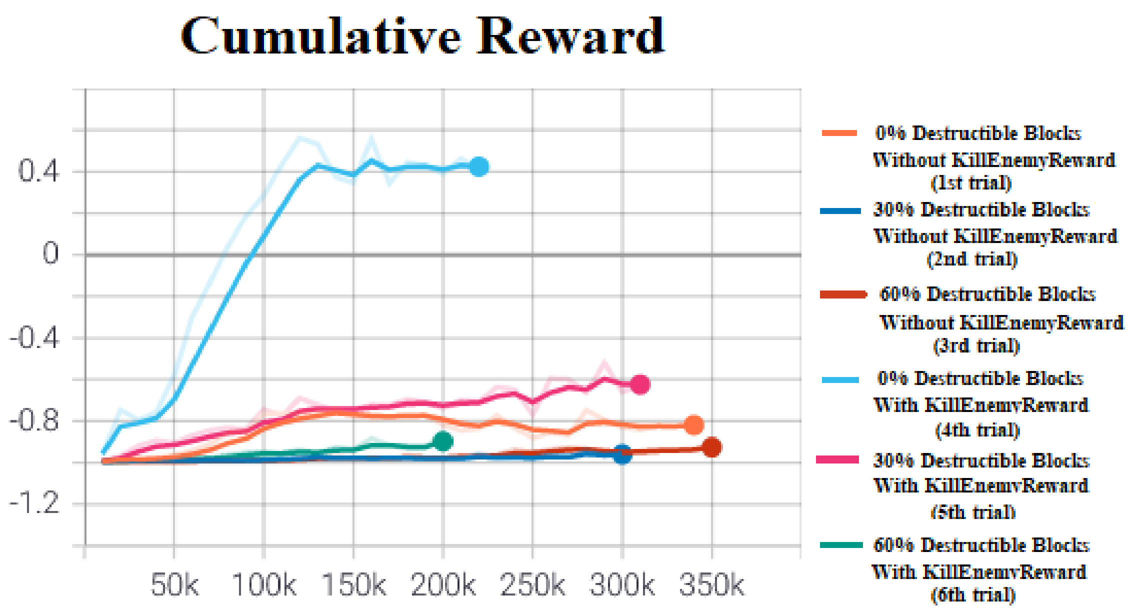

Figure 3.7: Comparison between the Trials of 1st and 2nd Batch of 2nd Series

In the end of the 2nd series of trials, the goal of studying how the environment configuration affects the learning process was achieved. It became apparent by the results that the less percentage of destructible blocks in the environment, the better the performance of the agent. However, the obtained results were still far from optimal.

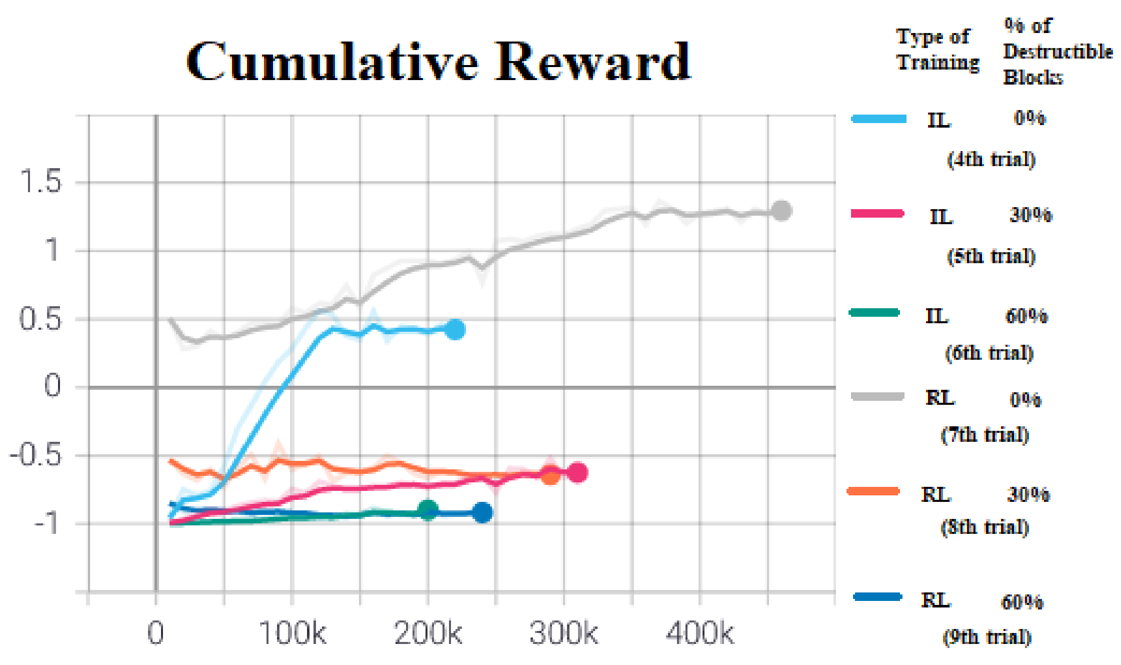

Figure 3.8: Results of The Last Batch of Trials of 2nd Series

3rd Series of Trials - A New Challenger Approaches: ML Agent vs Planning Agent

The third series of trials marks the introduction of the planning synthetic player in the training process. After the planning agent's development was completed, it began to be used as one of the opponents of the machine learning agent. A new demo comprising 100 matches played against the planning synthetic player and two idle ones was provided.

All of the trials of this series presented the rewards listed in Table 3.7.

Reward	Value
Winning the game	1.0
Dying	-1.0
Killing an enemy	0.75

Table 3.7: Rewards Used in Every Trial of 3rd Series

The first trial focused on performing imitation learning with both BC and GAIL, so the agent can present similar behaviors from those seen in the new demo. The environment configuration and the most relevant parameters for this first trial can be found in tables 3.8. The rest of the hyperparameters were the ones of 3.6.2.

Parameters	Values
% of Destructible Blocks	0%
Opponents	1 Planning and 2 Idle
Provided Demo	100 Episodes in an 0% destructible blocks environment
BC Strength	1
GAIL Strength	0.5

Table 3.8: Environment Configuration of 1st Trial of 3rd Series

The second trial focused on using reinforcement learning to improve the result of the trained model of the first trial. Therefore, the environment configuration was the same, with the exception of BC and GAIL being reduced to 0.01.

Results of these two trials can be visualized in Figure 3.9.

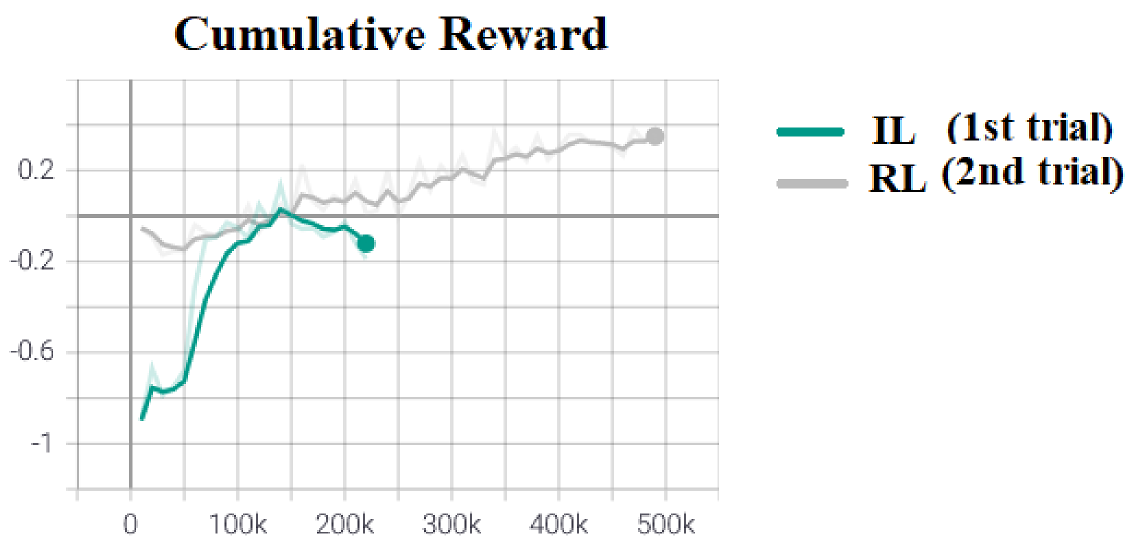

Figure 3.9: Results of The First Two Trials of 3rd Series

The next trial (third trial of the series) improved upon the previous reinforcement learning trial (second trial), with the aim of training the agent in a slightly harder scenario. BC and GAIL strength remained reduced to 0.01 so the agent could learn via extrinsic rewards.

The environment configuration was the same, with the exception of the training process happening in a game scenario with 10% destructible blocks.

Figure 3.10 shows the comparison between: 1) the first trial executed in this series, that consists of imitation learning in a 0% destructible blocks environment; 2) the second trial performed in this series, which is reinforcement learning upon the first trial, still in a 0% destructible blocks environment; 3) the third trial, which is reinforcement learning upon the second trial, this time in a 10% destructible blocks environment.

The last two trials of the series are analogous with the first two, since they focus on first training the agent with IL and then improving the model with RL's chosen algorithm, PPO.

Therefore, one trial was performed in a 20% destructible blocks environment, with the setup configuration of table 3.9.

Parameters	Values
% of Destructible Blocks	20%
Opponents	1 Planning and 2 Idle
Provided Demo	100 Episodes in an 30% destructible blocks environment
BC Strength	1
GAIL Strength	0.5

Table 3.9: Setup Configuration of The Last Two Trials of 3rd Series

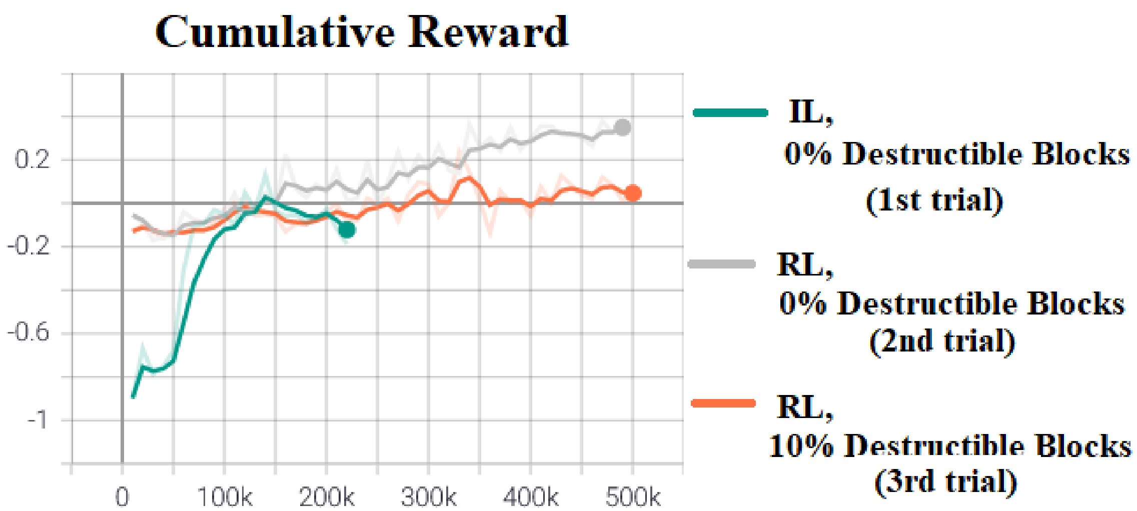

Figure 3.10: Comparison Between The First Three Trials of 3rd Series

Then, the last executed trial, which focuses on improving the previous one through RL, presents the same environment configuration, with the exception of both BC and GAIL strength being lowered to 0.01.

Results of these last two trials can be seen in Figure 3.11.

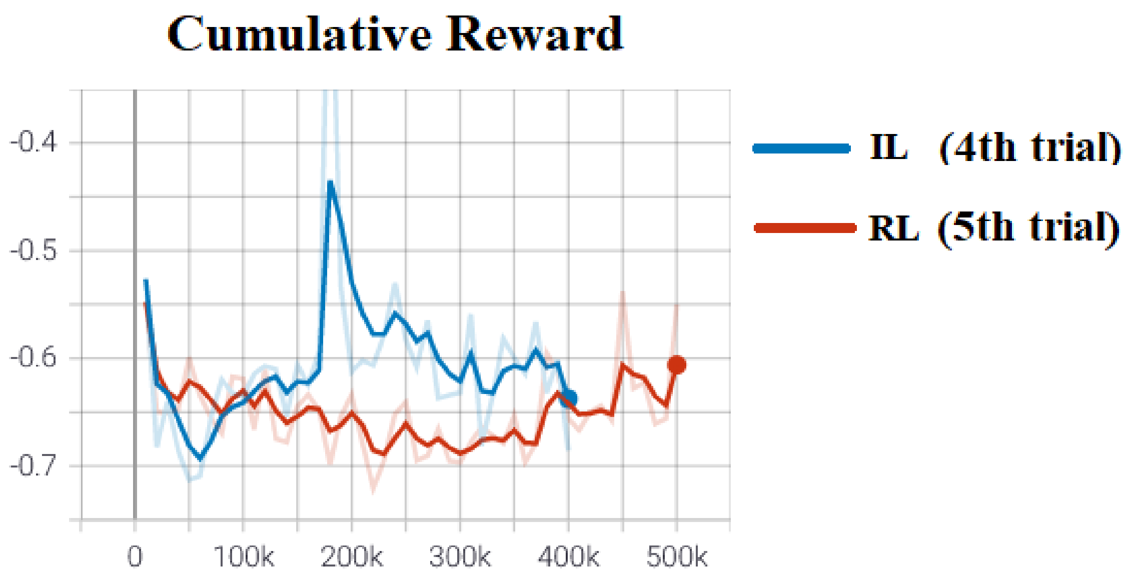

Figure 3.11: Comparison Between The First Three Trials of 3rd Series

One major achievement with this series of trials was the incorporation of the planning synthetic player in the training process. However, the results obtained remained modest and possible to be improved.

4th Series of Trials - Changing The Neural Network Architecture and Hyperparameters

The fourth series had the main goal of tweaking the agent's neural network architecture and its hyperparameters in order to get better results A brief meeting with Professor Doctor and Dissertation's Judge Ernesto Costa provided some helpful recommendations to help to guide this series of trials. In total, 12 trials were made during this series.

The rewards of Table 3.10 were the ones used in every trial of this series, except for the 10th trial.

Reward	Value
Winning the game	1.0
Dying	-1.0
Killing an enemy	0.75

Table 3.10: Rewards Used in Every Trial of the 4th Series, Except the 10th Trial

The first trial's aim was to test a new configuration of the neural network, increasing the amount of hidden layers from 2 to 4 and decreasing the number of units per hidden layer from 128 to 64. This way, the neural network is narrower but deeper. IL was done in an 0% destructible blocks environment. The setup configuration can be seen in table 3.11. The rest of the hyperparameters remain unchanged, being the ones of 3.6.2.

Parameters	Values
% of Destructible Blocks	0%
Opponents	1 Planning and 2 Idle
Provided Demo	100 Episodes in an 0% destructible blocks environment
BC Strength	0.5
GAIL Strength	0.5
Number of Hidden Layers	4
Number of Hidden Units	64

Table 3.11: Setup Configuration of The 1st Trial of 4th Series

Then, a second trial was performed with the exact same configuration, to improve upon the first via RL. Therefore, BC and GAIL strength was reduced to 0.01.

The results of both trials can be seen in Figure 3.12.

Then, two more trials were performed, in an analogous manner to the first two, but this time in a 30% destructible blocks environment.

Comparison between the four trials can be seen below in Figure 3.13.

The next pair of trials focused on testing new values for the number of epochs, batch size and buffer size hyperparameters.

The first one (fifth trial of the series in total) presented higher batch size and number of epochs than previous experiments. Its setup is presented in table 3.12.

The next trial (sixth trial of the series) reverted the batch size and number of epochs back to the most common values used so far, and tested an inferior buffer size as well. The setup was similar to the previous one, with just three differences, depicted in Table 3.13.

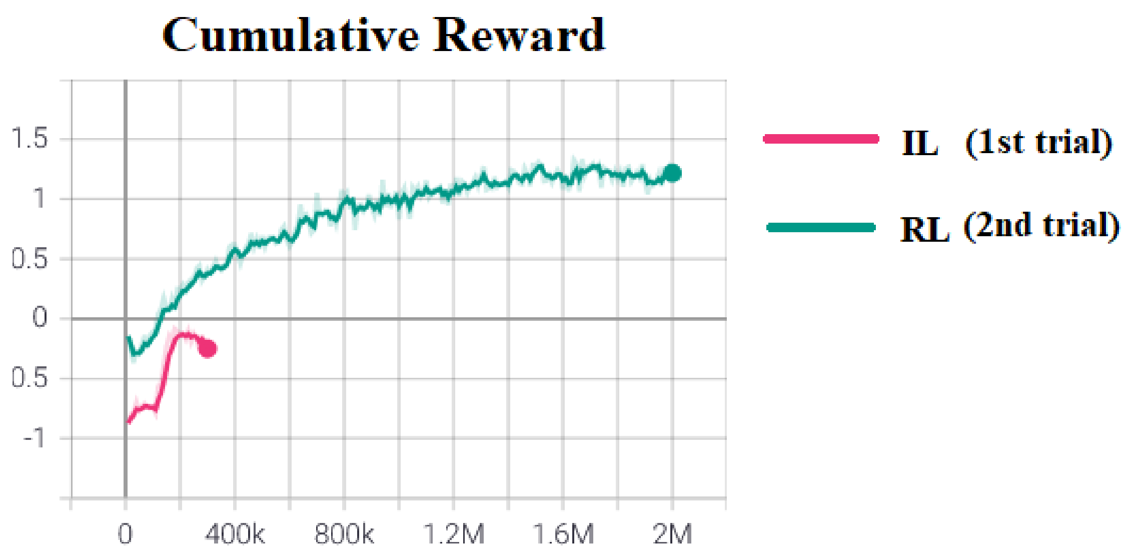

Figure 3.12: Results of The First Two Trials of 4th Series

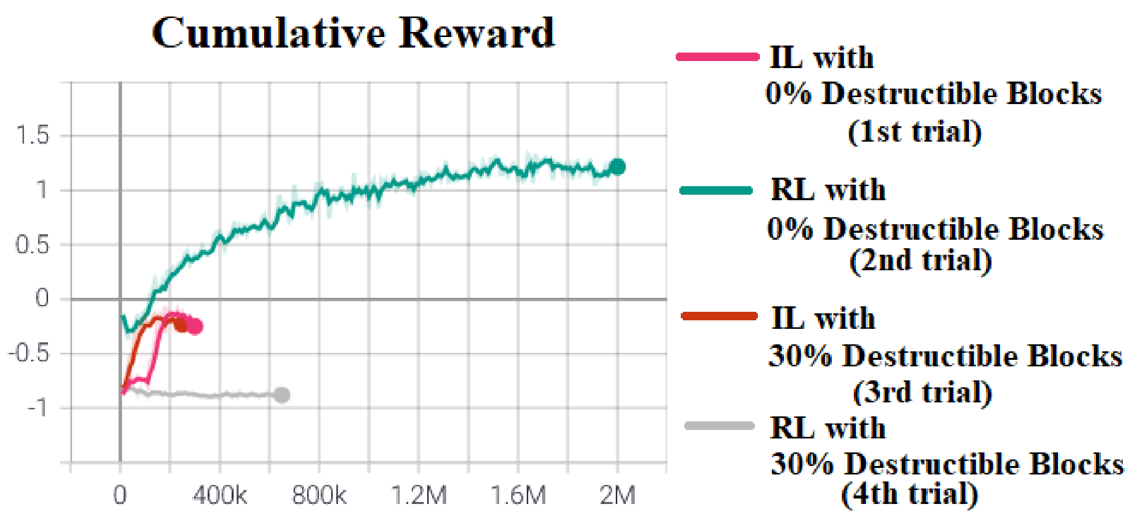

Figure 3.13: Comparison Between The First Four Trials of 4th Series

Parameters	Values
% of Destructible Blocks	0%
Opponents	1 Planning and 2 Idle
Provided Demo	100 Episodes in an 0% destructible blocks environment
BC Strength	0.01
GAIL Strength	0.01
Number of Hidden Layers	4
Number of Hidden Units	64
Batch Size	512
Number of Epochs	5

Table 3.12: Setup Configuration of The 5th Trial of 4th Series

Hyperparameter	Value
Batch Size	256
Number of Epochs	3
Buffer Size	2048

Table 3.13: Changes in Hyperparameters from 5th to 6h Trial of 4th Series

The comparison between these two new trials and the second trial of the series can be seen below in Figure 3.14.

Since a buffer size of 2048 reached the same performance as one of 10240 in less time, 2048 was chosen as the default value for buffer size. So, from this point onwards, every machine learning run presents a buffer size of 2048 unless stated otherwise.

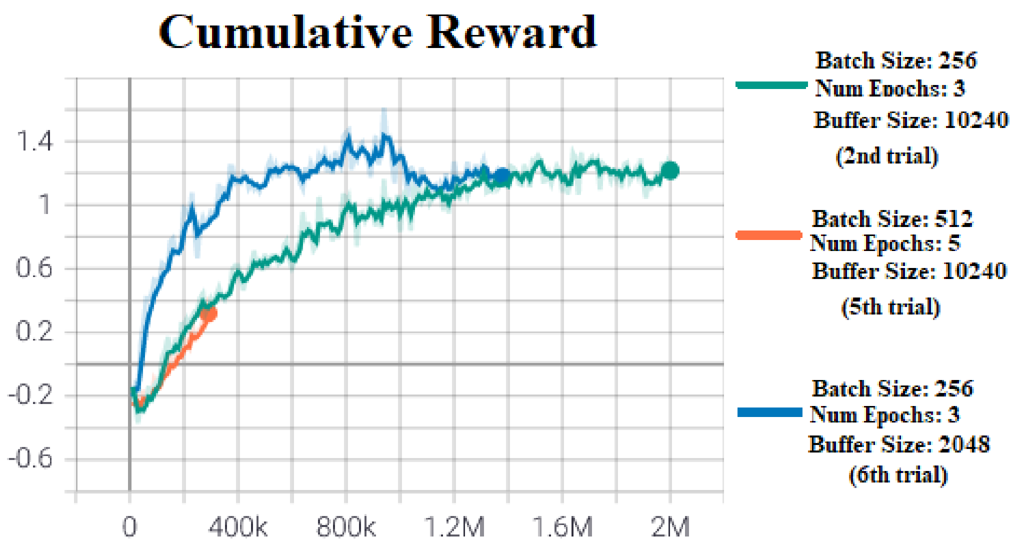

Figure 3.14: Comparison Between The 2nd, 5th and 6th Trials of 4th Series

The previous executed trial (6th one), with buffer size 2048 was selected to be the starting point for the next two trials. In fact, RL was applied to the last trial's obtained brain model, but for environments with different percentages of destructible blocks.

One of the trials happened in a 30% destructible blocks environment, and the other in a 10% scenario. The setup for these two trials is the depicted in Table 3.14.

Parameters	Values
% of Destructible Blocks	30% for one trial, 10% for another
Opponents	1 Planning and 2 Idle
BC Strength	0.01
GAIL Strength	0.01
Number of Hidden Layers	4
Number of Hidden Units	64
Batch Size	256
Number of Epochs	3
Buffer Size	2048

Table 3.14: Setup Configuration of The 7th and 8th Trials of 4th Series

Comparison between the three referred trials (6th, 7th and 8th trials) can be analyzed in Figure 3.15.

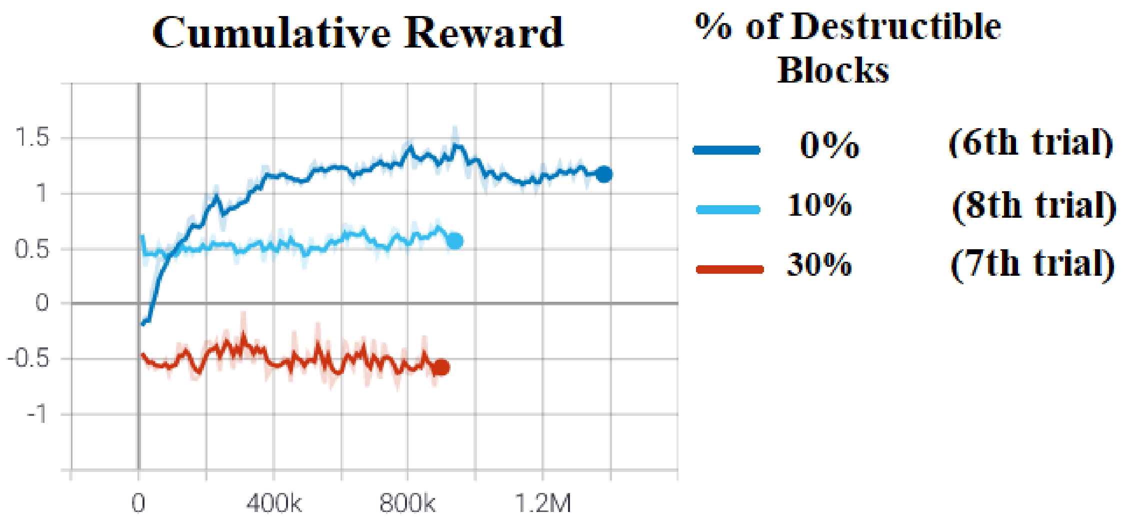

Figure 3.15: Comparison Between The 6th, 7th and 8th Trials of 4th Series

The next trial's main goal was to analyse the discount factor's (γ) impact in the learning process. Henceforth, the new trial (9th trial of the series) presented a γ of 0.85, lower than the 0.99 value used so far. The rest of the environment's and neural network's configuration remained the same as the previous trials.

The next graphic 3.16 shows the comparison between the new trial, with γ 0.85, and a previous trial executed (the 6th one) in the exact same conditions, with γ 0.99.

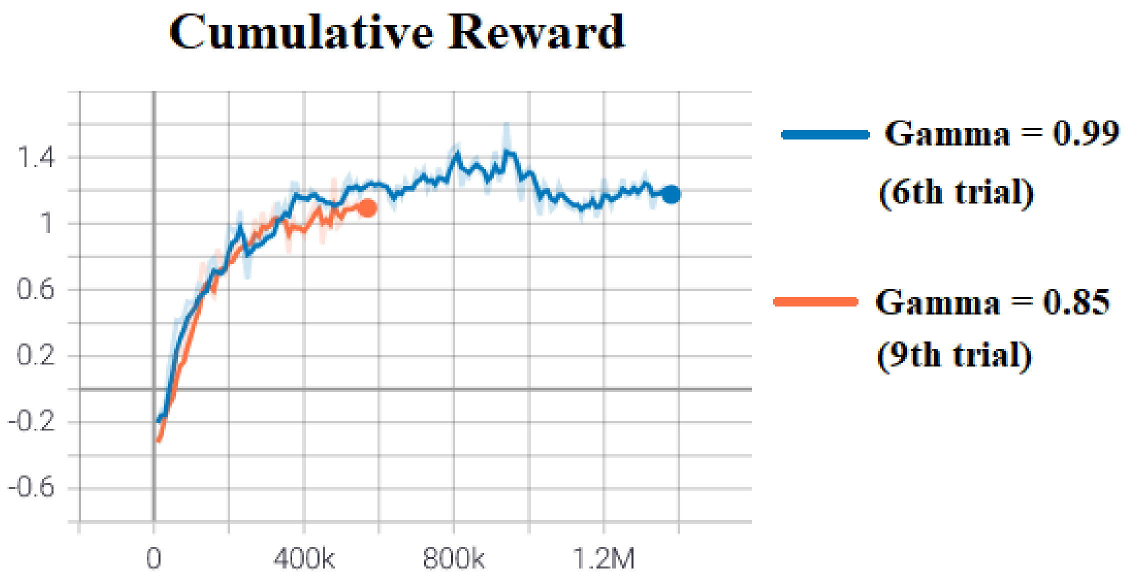

Figure 3.16: Comparison Between Trials with Different γ (6th and 9th of 4th Series)

Next parameter to be tested is the value of the reward for killing an enemy. Until this point, a value of 0.75 was used. A new trial was made to evaluate the impact of decreasing this reward value to 0.5. Its setup is the same of previous trials, and can be consulted in table 3.15. The rewards table is the one of 3.16.

Parameters	Values
% of Destructible Blocks	0%
Opponents	1 Planning and 2 Idle
Provided Demo	100 Episodes in an 0% destructible blocks environment
BC Strength	0.01
GAIL Strength	0.01
Number of Hidden Layers	4
Number of Hidden Units	64
Batch Size	256
Number of Epochs	3
Buffer Size	2048
Gamma	0.85

Table 3.15: Setup Configuration of The 7th and 8th Trials of 4th Series

Reward	Value
Winning the game	1.0
Dying	-1.0
Killing an enemy	0.5

Table 3.16: Rewards Used in The 10th Trial of 4th Series

The results of the new trial and the previous one are in Figure 3.17, to compare the different values for the reward of killing an enemy.

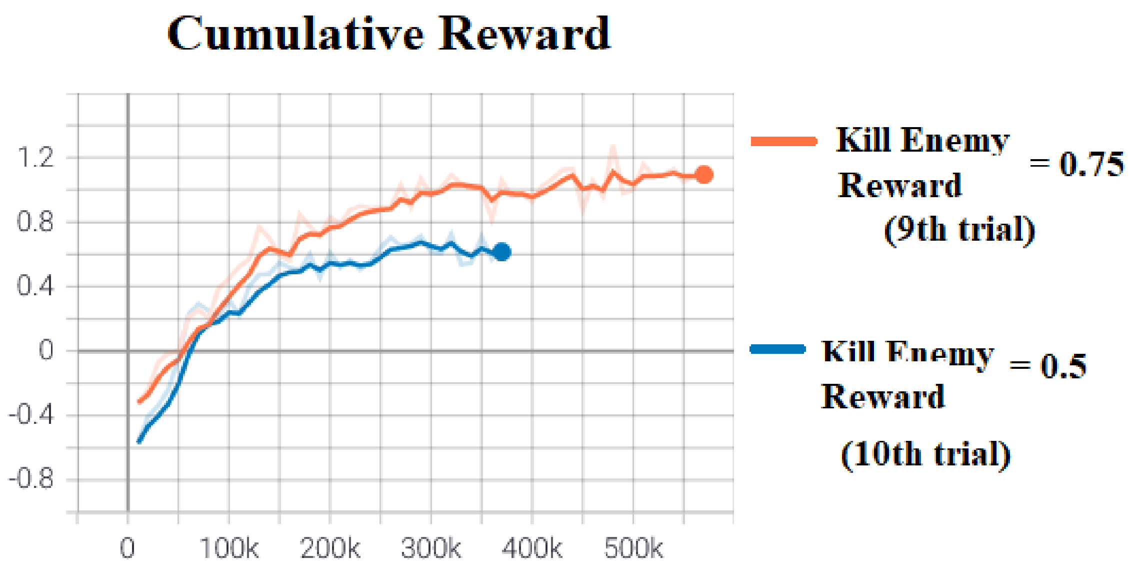

Figure 3.17: Comparison Between Trials with Different Values for The Reward of Killing an Enemy (9th and 10th of 4th Series)

The two last trials of this series were the firsts to incorporate self-play, in which the machine learning agent faces itself during the learning process. The first was made in a 0% destructible blocks environment, and the second one in a 30% environment. The environment configuration can be seen in Table 3.17.

Parameters	Values
% of Destructible Blocks	0% one trial, 30% the other
Opponents	1 Planning and 1 Machine Learning (previous iterations of itself during the training process)
Provided Demo	100 Episodes in an 0% destructible blocks environment
BC Strength	0.01
GAIL Strength	0.01
Number of Hidden Layers	4
Number of Hidden Units	64
Batch Size	256
Number of Epochs	3
Buffer Size	2048
Gamma	0.85

Table 3.17: Setup Configuration of The Last Two Trials of 4th Series

With the incorporation of self-play, the following lines must be added to the configuration file:

```
self_play:
  window: 10
  play_against_latest_model_ratio: 0.5
  save_steps: 50000
  swap_steps: 2000
  team_change: 100000
```

The self-play parameters are the following:

- *window*: Size of the window used to save past snapshots of the agent's policy. A value of 10 will make sure that the last 10 previous snapshots are saved. Whenever a new policy snapshot is saved, the oldest one in the window is discarded.
- *play_against_latest_model_ratio*: Probability for the agent to play against an opponent with the latest saved policy snapshot or against one of the others in the window, chosen randomly.
- *save_steps*: Number of *trainer* steps before saving a new snapshot of the agent's policy. *Trainer* steps are equal to environment steps multiplied by the number of learning agents in the environment.
- *swap_steps*: Number of *ghost* steps before changing the policy snapshot of the opponent. *Ghost* steps are the steps taken by the agent's opponent, that has a fixed policy and is not learning.
- *team_change*: Number of *trainer* steps before swaping teams: the agent becomes the opponent, with a fixed policy, and the opponent becomes the learning agent.

These parameters were never changed in any circumstance, including in the next series of trials, that also uses self-play. Therefore, whenever a trial is being described as incorporating self-play, one can assume that these are the values for each parameter, even in other series of trials besides this one.

Results of both trials can be compared in Figure 3.18.

Changing the neural network architecture and hyperparameters revealed beneficial, with Trials 2, 6 and 9 obtaining the best results so far.

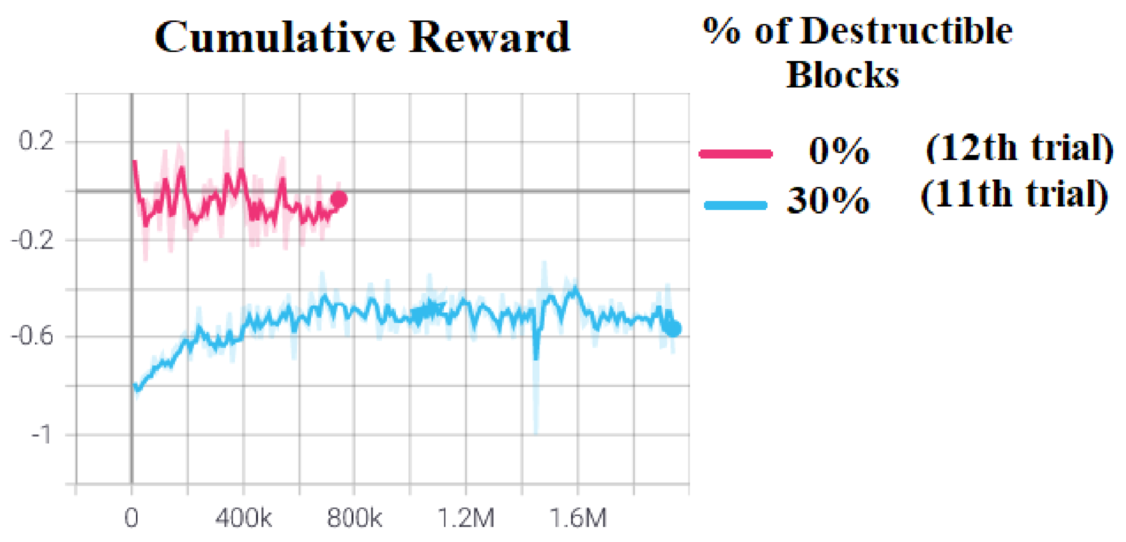

Figure 3.18: Comparison Between The Last Two Trials of 4th Series

5th Series of Trials - Make or Break: Last Trials

One last series of trials was done in order to try to optimize the results. This series is characterized by incorporating self-play in every trial and the inclusion of a curiosity module. All of the trials used the same demo, a set of 50 matches played by a human player against a planning synthetic player and a machine learning agent in a 30% destructible blocks environment. A total of 11 trials were performed in this series. Most of the the values for the hyperparameters are adapted from [França et al., 2019].

The first two trials used BC coupled with the curiosity module and self-play in order to make the agent learn to mimic behaviors seen in the demo.

All of the trials of this series had the rewards listed in Table 3.18.

Reward	Value
Winning the game	1.0
Dying	-1.0
Killing an enemy	0.75
Iteration Penalty	-0.0001

Table 3.18: Rewards Used in Every Trial of 5th Series

Curiosity is added to the configuration file under the *reward_signals* section. There The bottom of the configuration file now looks like the following:

```
reward_signals:
  extrinsic:
    gamma: 0.99
    strength: 1.0
  curiosity:
    strength: 0.1
  gail:
    strength: 0.01
    demo_path: Demos/[]name of Demo].demo
```

```
behavioral_cloning:
  strength: 0.01
  demo_path: Demos/[name of Demo].demo
self_play:
  window: 10
  play_against_latest_model_ratio: 0.5
  save_steps: 50000
  swap_steps: 2000
  team_change: 100000
```

The configuration of the two trials is listed in Table 3.19, alongside the default self-play parameters.

Parameters	Values
% of Destructible Blocks	0% one trial, 30% the other
Opponents	1 Planning and 1 Machine Learning
BC Strength	1.0
GAIL Strength	0.01
Curiosity Strength	0.1
Number of Hidden Layers	3
Number of Hidden Units	128
Batch Size	128
Gamma	0.99

Table 3.19: Setup Configuration of The First Two Trials of 5th Series

The results are presented in Figure 3.19.

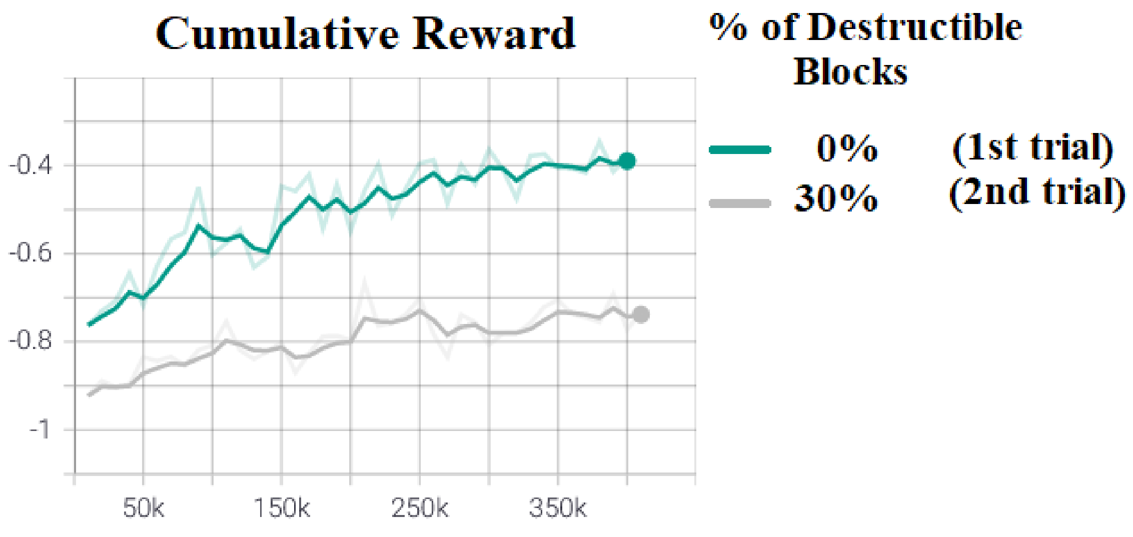

Figure 3.19: Results of The First Two Trials of 5th Series

The next two trials aimed at improving the performance of the first two. Therefore, BC strength was reduced to 0.01, while the rest of the parameters remained the same.

Comparison between the four trials can be seen in Figure 3.20.

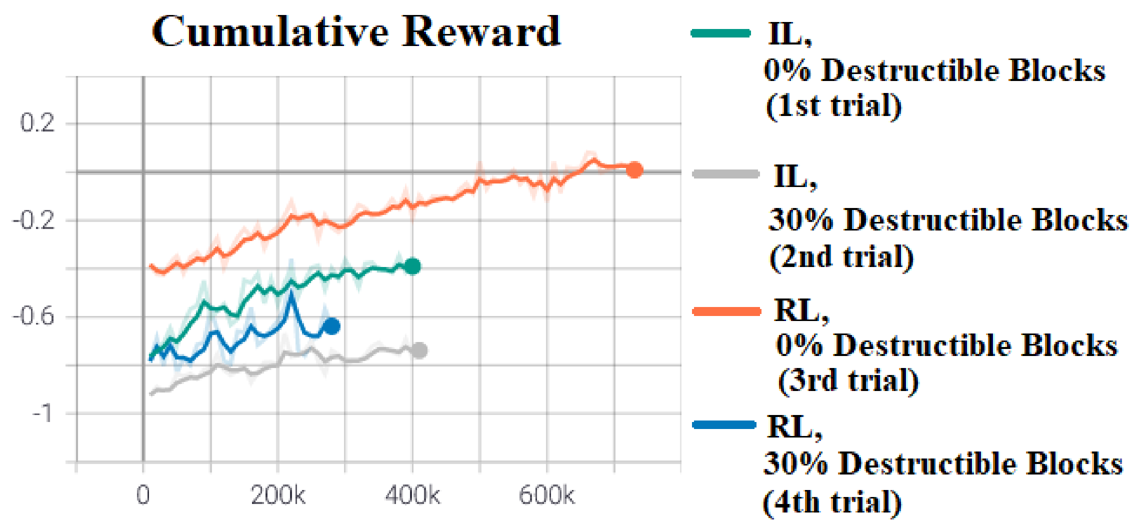

Figure 3.20: Comparison Between The First Four Trials of 5th Series

A new trial (fith of the series in total) was performed to improve the obtained result of one of the last reinforcement learning trials. More precisely, the previous third trial of this series, in which RL is performed in an environment with no destructible blocks, is used as a baseline for the new trial. However, this new run happened in an environment with 30% destructible blocks. The configuration is exactly the same as the one of previous trials.

Figure 3.21 shows the comparison between the new trial (in red), the trial that acted as baseline (in orange) and the fourth trial of this series (in blue), in which RL was performed upon the second trial that used IL in a 30% destructible blocks configuration.

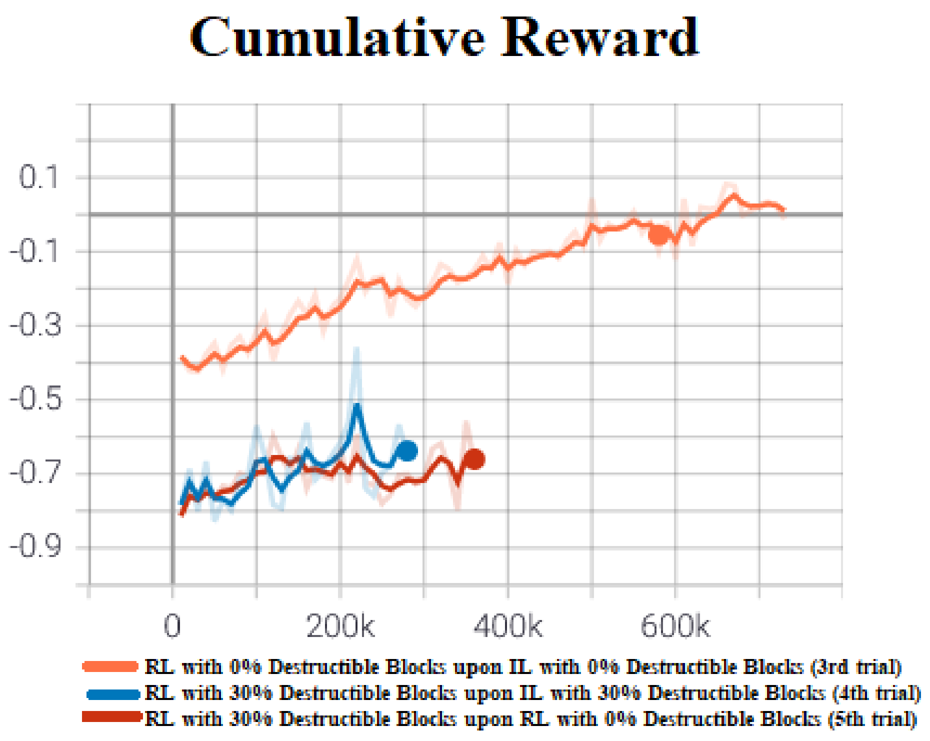

Figure 3.21: Comparison Between The 3rd, 4th and 5th trials of 5th Series

The next two trials (6th and 7th overall) are very similar to the first two trials of this series. The only difference is that IL is performed by coupling BC and GAIL in the two new trials. One of the trials is performed in an environment with no destructible blocks, and the other in an scenario with 30% of that type of blocks. Therefore, the configuration for these two new trials is the same as the one used for the first two, with the exception that GAIL strength is increased to 0.5. Comparison between the two new trials and the first two can be seen in Figure 3.22.

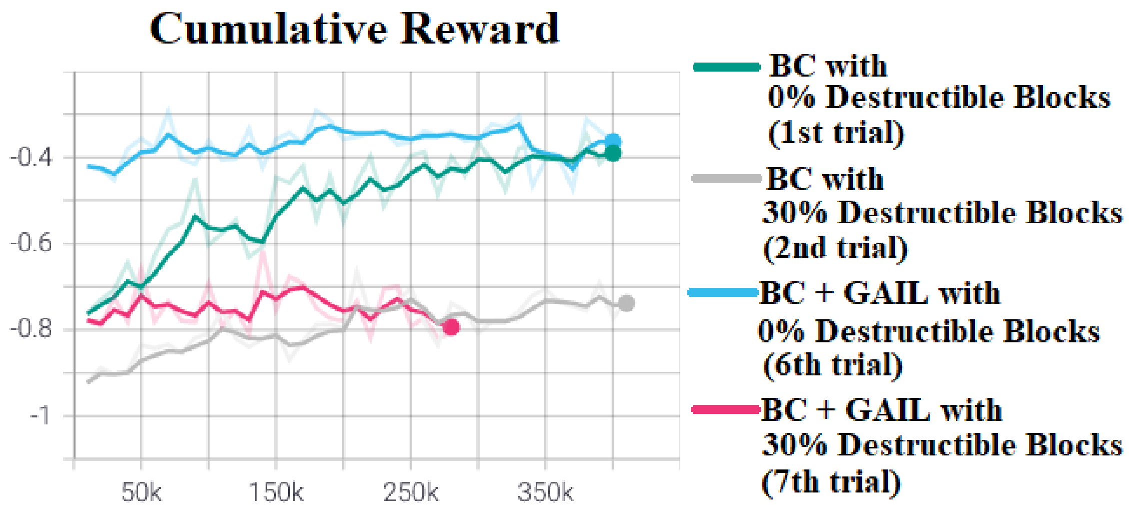

Figure 3.22: Comparison Between The First Two Trials and the 6th and 7th Trials of 5th Series

The next two trials (8th and 9th overall) executed RL to improve the results of the previous two trials (6th and 7th). So both BC and GAIL strength is decreased to 0.01, in order for the agent to learn more via the environment's rewards.

Comparison between the two new trials and the previous two can be seen in Figure 3.23.

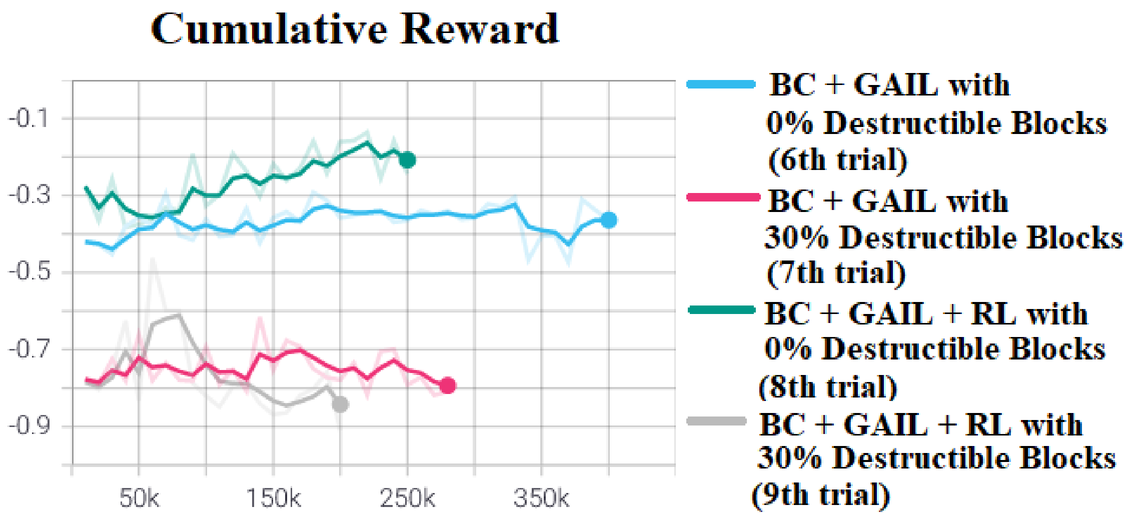

Figure 3.23: Comparison Between The 6th, 7th, 8th and 9th Trials of 5th Series

One of the last trials of this series incorporated a memory component in the training process. The configuration mirrors the first trial of the series, that is depicted in Table 3.19. The memory component parameters are listed in Table 3.20.

Parameter	Value
Memory Size	256
Sequence Length	32

Table 3.20: Memory Parameters of The Last Two Trials of 5th Series

The comparison between this trial with memory and a previous trial without it executed in the same conditions (the first trial of the series) can be seen in Figure 3.24.

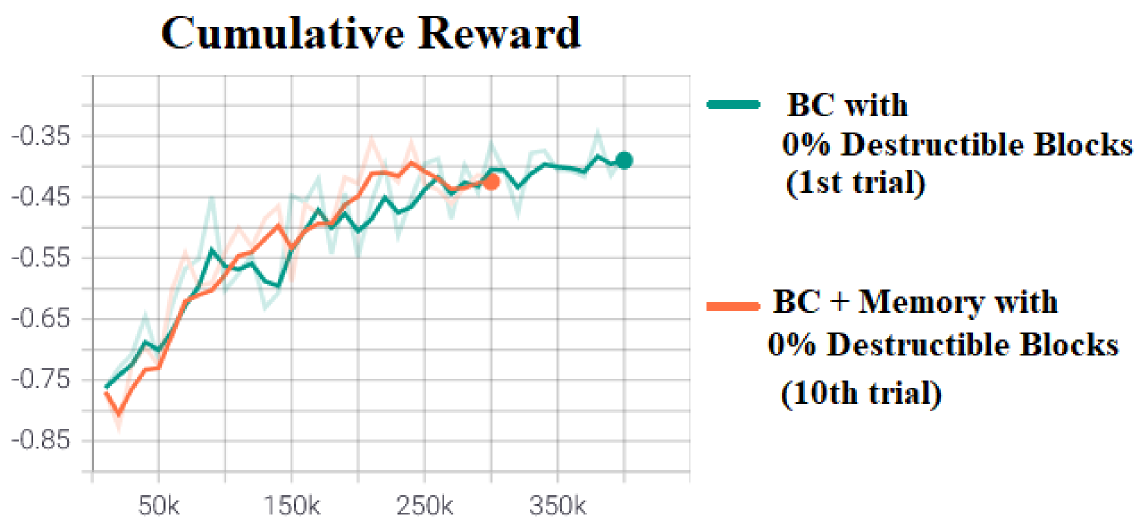

Figure 3.24: Comparison Between A Trial With (10th) and a Trial Without (1st) Memory Component, of 5th Series

The last trial of the series is the application of RL on the previous memory trial (10th). Therefore the configuration is the same as the previous trial, except that BC strength is decreased to 0.01.

The comparison between the two trials involving the memory component can be observed in Figure 3.25.

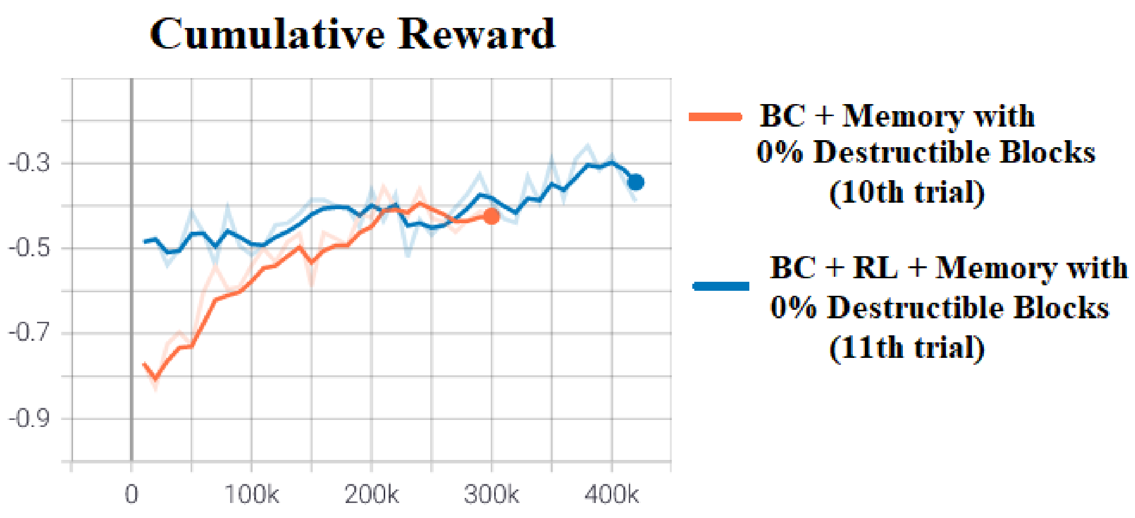

Figure 3.25: Comparison Between The Two Last Trials of 5th Series, that Involve a Memory Component

This series was not able to get better results than the ones obtained in fourth series, despite the incorporation of a curiosity module and a strong reliance on self-play. The last two trials, that incorporated a memory component, did not produce relevant results. This may be due to a wrong setup of framework parameters, or lack of training time.

This page is intentionally left blank.

Chapter 4

Experiments and Evaluation

This chapter shows the experiments done in order to validate the potential use of synthetic players as players and testers of game scenarios generated via Procedural Content Generation (PCG), and discusses the obtained results.

4.1 Selection of Machine Learning Agents

Before the final evaluation phase to measure the synthetic players compliance to the three pre-established requirements, some pre-experiments needed to be made in order to select the best machine agents to use.

In order to fulfill requirements #2 and #3, a synthetic player must fulfill requirement #1, since in order to play in new scenarios and to act in a believable way, the agent must be able to play and win the game first. Therefore, a series of small tournaments composed of 1000 games were made in order to determine which are the best machine learning agents. Each tournament pits against each other two different machine learning agents, with a planning synthetic player to the mix. The machine learning agent with more victories (regardless if the planning synthetic player has more victories or not) over the other advances to a next tournament, while the machine learning synthetic player with less victories is removed from the experiments. With this method, eventually the two best machine learning agents for a particular environment configuration are discovered and selected. In total, 4 machine learning synthetic players will be selected for the final evaluation phase: the two best that were trained in environments with no destructible blocks, and the two best that were trained in a 30% destructible blocks environment.

The results of all tournaments can be consulted at once in Table 4.1. For brevity reasons, the trials in the table are referred as *[number of series of trials]-[number of trial]*. For instance, to address the 6th trial of 4th series, its definition is 4-6. A result cell shows a score obtained in a tournament between the two trials. For instance, 16 - 12 means that the corresponding trial of the first column of the table obtained a 16% of victories in 1000 matches while facing the other machhine learning agent, that obtained 12%. If a cell of the table is blank, means that a tournament between the two respective agents did not happen.

From all the tournaments organized, four machine learning agents proved to be better than the remaining. First, the 6th and 9th trial of the 4th series were selected as the best machine learning agents that were trained in an environment with no destructible blocks.

Trials	2-5	2-7	4-2	4-6	4-7	4-9	4-11	5-3
2-5	N/A				0.9-16			
2-7		N/A				1.5 - 9.8		
4-2			N/A			6.4 - 8.4		
4-6				N/A	5.5-15.1	8 - 7.1	6.3-12.8	
4-7	16-0.9			15.1-5.5	N/A	16.8 - 4	12.2-10.1	
4-9	9.8-1.5	8.4-6.4	7.1-8	4-16.8		N/A	4.8-13.8	8.3-5.7
4-11				10.8-6.3	10.1-12.2	13.8 - 4.8	N/A	
5-3						5.7 - 8.3		N/A

Table 4.1: Results of Tournaments to Select The Best Machine Learning Agents. N.A: Non-Applicable

Then, trials 7th and 11th of the 4th series were selected as the top two machine learning agents trained in an environment configuration of 30% destructible blocks. Henceforth, these four synthetic players will be the ones used in the final evaluation phase.

4.2 Requirement #1: Ability to Play and Win The Game

For this requirement, the experiments were done in the form of tournaments, where the synthetic players play against each other as well as against a human player. The experiments were repeated for multiple game scenario configurations of the previously described case study, i.e, for different starting percentages of destructible blocks on the map.

4.2.1 For 0% of starting destructible blocks

This is the easiest possible configuration. With the absence of destructible blocks, the map is always fixed, with synthetic players having a clear path for every opponent.

Tournament 1

The first tournament was composed of 10000 matches, with the players being:

- Planning synthetic player of 3.4;
- Machine Learning synthetic player obtained from 6th trial of 4th series of trials, described in 3.6.3;
- Machine Learning synthetic player obtained from 9th trial of 4th series of trials, described in 3.6.3;

The tournaments results are in Table 4.2.

Tournament 2

The second tournament was composed of 100 matches, with the players being the same as the ones of tournament 1 plus the addition of a human player.

Its results can be consulted in Table 4.3.

Player	Win Percentage
Planning	50.59%
Machine Learning (6th Trial of 4th Series)	17.39%
Machine Learning (9th Trial of 4th Series)	13.94%
Draw	18.08%

Table 4.2: Results of Tournament 1 of Requirement #1

Player	Win Percentage
Human Player	45%
Planning	21%
Machine Learning (6th Trial of 4th Series)	3%
Machine Learning (9th Trial of 4th Series)	2%
Draw	29%

Table 4.3: Results of Tournament 2 of Requirement #1

4.2.2 For 30% of starting destructible blocks

This configuration is harder than the previous one. Now with the presence of destructible blocks, sometimes players have their path obstructed, and have to strategically place some bombs to carve their way to the enemies.

Tournament 3

Tournament 3 followed the same principles of tournament 1, with 10000 matches being played, pitting the following three modeled synthetic players against each other:

- Planning synthetic player of 3.4;
- Machine Learning synthetic player obtained from 7th trial of 4th series of trials, described in 3.6.3;
- Machine Learning synthetic player obtained from 11th trial of 4th series of trials, described in 3.6.3;

The tournament results are in Table 4.4.

Player	Win Percentage
Planning	59.98%
Machine Learning (7th Trial of 4th Series)	10.91%
Machine Learning (11th Trial of 4th Series)	9.33%
Draw	19.78%

Table 4.4: Results of Tournament 3 of Requirement #1

Tournament 4

Tournament 4 presented the same configuration of tournament 2, in which 100 matches were played. The players are the same of tournament 3, with the addition of a human player.

The results of the tournament can be consulted in Table 4.5

Player	Win Percentage
Human Player	59%
Planning	15%
Machine Learning (7th Trial of 4th Series)	5%
Machine Learning (11th Trial of 4th Series)	2%
Draw	19%

Table 4.5: Results of Tournament 4 of Requirement #1

4.3 Requirement #2: Ability to Play New Untested Game Scenarios

In order to test the compliance of the synthetic players with this requirement, a series of tournaments were done, like the tournaments for requirement #1. However, in this case, only map configurations with 20% and 60% of destructible blocks were tested. The machine learning agents were never trained in such map configurations (20% and 60%), so they experienced these game scenarios for the first time. The planning synthetic player was also never tested in such scenarios.

4.3.1 For 20% of starting destructible blocks

This configuration resembles the 30% configuration previously tested in requirement #1, however slightly easier.

Tournament 1

This tournament pitted the following three synthetic players against each other in a series of 10 000 matches:

- Planning synthetic player of 3.4;
- Machine Learning synthetic player obtained from 6th trial of 4th series of trials, described in 3.6.3;
- Machine Learning synthetic player obtained from 7th trial of 4th series of trials, described in 3.6.3;

The tournament results can be analysed in Table 4.6.

Player	Win Percentage
Planning	60.59%
Machine Learning (6th Trial of 4th Series)	9.34%
Machine Learning (7th Trial of 4th Series)	14.91%
Draw	15.16%

Table 4.6: Results of Tournament 1 of Requirement #2

Tournament 2

This tournament gathered all the three players that participated in Tournament 1 plus a human player, and 100 matches were played.

Its results are depicted in Table 4.7.

Player	Win Percentage
Human Player	66%
Planning	7%
Machine Learning (6th Trial of 4th Series)	3%
Machine Learning (7th Trial of 4th Series)	4%
Draw	20%

Table 4.7: Results of Tournament 2 of Requirement #2

4.3.2 For 60% of starting destructible blocks

This is the hardest configuration and the one considered as the closest depiction of an original *Bomberman* game scenario. There is an abundance of destructible blocks on the map, and players often start in a place with little room to maneuver. They have to be careful and to place bombs in the right places, in order to get more freedom of movement, and to eventually find their way to their enemies.

Tournament 3

Like Tournament 1, the three modelled synthetic players faced each other throughout 10000 matches.

The tournament results are listed in Table 4.8.

Tournament 4

Tournament 4 is similar to Tournament 2, in which all the four players played against each other in a series of 100 matches.

The results are in Table 4.9.

Player	Win Percentage
Planning	64.77%
Machine Learning (6th Trial of 4th Series)	1.18%
Machine Learning (7th Trial of 4th Series)	8.92%
Draw	25.13%

Table 4.8: Results of Tournament 3 of Requirement #2

Player	Win Percentage
Human Player	62%
Planning	11%
Machine Learning (6th Trial of 4th Series)	0%
Machine Learning (7th Trial of 4th Series)	3%
Draw	24%

Table 4.9: Results of Tournament 4 of Requirement #2

4.4 Requirement #3: Ability to Play in a Believable Way

Playing in a believable way is a subjective concept. Like stated in 2.3.2, the most common way to evaluate this is by doing a Turing test. However, due to time constraints, it was infeasible to gather feedback from a statistically relevant number of people. Therefore, another way to assert this requirement was devised. Several behavioral traits categorized as believable were defined. A series of four videos showing gameplay from each synthetic player and also from a human player were shown to a small number of people, who were inquired about the presence or absence of the pre-defined behavioral traits by the players in the videos. Nonetheless, every participant was also asked to identify in the videos which players are human and which are not. This was mostly done in the name of curiosity since, as already said, the total number of gathered samples is not statistically enough to execute a real Turing test. The videos used in the survey are stored at

`https://github.com/Drca1997/Tese/tree/Diogo/Tese/Assets/Survey`

In the first video, the four players are:

- the planning synthetic player of section 3.4 (green player);
- the machine learning agent of the 6th trial of 4th series of trials (pink player);
- the machine learning agent of the 9th trial of 4th series of trials (blue player);
- a synthetic player that executes a random action each turn (orange player);

In the second video, the four players are the same as the ones in the previous video, with the exception of the machine learning agent of the 9th trial of 4th series being replaced by the agent of the 7th trial of the same series (in the 2nd video, the colors of the players are: 1) planning agent: orange; 2) machine learning agent of the 6th trial of 4th series: green; 3) machine learning agent of the 7th trial of 4th series: pink; 4)random agent: blue).

Videos 3 and 4 incorporate a human player, that replaces the random synthetic player. Therefore, in the third video, the players are:

- a human player (pink player);
- the planning synthetic player of section 3.4 (blue player);
- the machine learning agent of the 6th trial of 4th series of trials (orange player);
- the machine learning agent of the 9th trial of 4th series of trials (green player);

In the fourth video, the four players are the same as the ones in the previous video, with the exception of the machine learning agent of the 9th trial of 4th series being replaced by the agent of the 7th trial of the same series (in the 4th video, the colors of the players are: 1) human player: green; 2) planning agent: pink; 3) machine learning agent of the 6th trial of 4th series: blue; 4) machine learning agent of the 7th trial of 4th series: orange).

Five people were asked to watch each video. Then, they were asked to classify each player in each video according to the following three attributes:

- **Apparent Rational Competence**: the apparent capacity of the agent to play the game. This translates into an apparent notion of the game rules, and not putting itself unnecessarily in precarious positions, like trapping itself with one of is own bombs.
- **Apparent Intentionality**: the clarity of intent behind the agent's actions. For instance, an agent that simply roams the map aimlessly would obtain the worst possible score in this attribute.
- **Apparent Anticipatory Behaviour**: ability of the agent to anticipate to certain situations and taking advantage of them, like trapping an enemy, or avoiding a danger zone.

Each attribute's score follows this scale:

- 0 - No evidences of;
- 1 - Some evidences of;
- 2 - Frequent evidences of;
- 3 - Always;

Table 4.10 compiles all the obtained classifications from the inquiries. Each cell corresponds to the number of times (out of the total number of ratings) a particular agent was graded with a particular score in a certain attribute in the inquiries. For instance, a 1/10 means that the player was rated with a particular score 1 out of 10 times.

Player	Apparent Rational Competence				Apparent Intentionality				Apparent Anticipatory Behaviour			
	0	1	2	3	0	1	2	3	0	1	2	3
Human	0/10	0/10	1/10	9/10	0/10	0/10	1/10	9/10	0/10	1/10	1/10	8/10
Planning	0/20	1/20	5/20	14/20	1/20	3/20	12/20	4/20	1/20	3/20	10/20	6/10
Machine Learning (6th trial of 4th series)	9/20	1/20	5/20	5/20	8/20	4/20	6/20	2/20	9/20	5/20	3/20	3/20
Machine Learning (7th trial of 4th series)	6/10	3/10	1/10	0/10	8/10	1/10	0/10	1/10	6/10	4/10	0/10	0/10
Machine Learning (9th trial of 4th series)	10/10	0/10	0/10	0/10	8/10	2/10	0/10	0/10	10/10	0/10	0/10	0/10
Random	3/10	6/10	1/10	0/10	3/10	5/10	2/10	0/10	6/10	3/10	1/10	0/10

Table 4.10: Results of Inquiries Regarding Requirement #3

In addition to the results of Table 4.10, another score was obtained, that refers to the number of times that each player was labeled as human. However, this score must be treated as a curiosity more than an official experiment result, since the low number of participants invalidates the execution of a real Turing test. The results for this attribute can be seen in Table 4.11.

Player	Number of Times Labelled as Human
Human Player	8 out of 10
Planning	3 out of 20
Machine Learning (6th Trial of 4th Series)	1 out of 20
Machine Learning (7th Trial of 4th Series)	1 out of 10
Machine Learning (9th Trial of 4th Series)	0 out of 10
Random	1 out of 10

Table 4.11: Number of Times that a Player was Labelled as Human in the Inquiries for Requirement #3

4.5 Discussion of Results

This section explains the meaning of the experiments' results, and provides some possible explanations for them.

4.5.1 Machine Learning Training Results

In total, five series of machine learning training trials were done. The series that proved to be the most fruitful was the fourth one, which was characterized by the tweaks done to the neural network's architecture and to its hyperparameters. All trials with the highest cumulative rewards (the 2nd, 6th and 9th trials of the series) had two features in common and first being the same neural network architecture: 4 hidden layers and 64 units per layer. These values differed from all the trials that had been executed before, in which the neural network was shallower, with only 2 hidden layers, and wider, with 128 units per layer. The second feature that these trials had in common was that they were all RL-based, which means that the strength of IL methods like BC and GAIL was reduced almost to 0, in order for the agent to improve its performance via the extrinsic rewards.

Besides the aforementioned trials of the 4th series, the rest of the trials did not produce relevant results, regardless of series. This could be due to several reasons:

- the agent's state representation was not the most appropriate one for the training process.
- the training runs needed more execution time.
- the difficulty of the rewards being delayed in the game. The agents only received positive rewards if they either won the game or killed an enemy, and in both cases, it required the agents to plant a bomb several turns prior to collecting the reward.

4.5.2 Results of Requirement #1

Regarding the first requirement, which is the ability of the synthetic player to play and win the game, results show a clear dominance of the planning synthetic player over all the other agents. In tournament 1, the planning agent can win half of the 10000, whereas no machine learning agent can reach a 20% of wins. In tournament 2, the difference between a human player and every synthetic player is noticeable. The planning agent registers a drastic decrease of win percentage, obtaining only 21% of them, which is a far cry from the 50% of the previous tournament, without a human player as opponent. All of the machine learning agents, once again, perform worse than the planning synthetic player, not being able to obtain higher than a 3% of wins.

Tournaments 3 and 4 happened within an environment with the presence of destructible blocks, and that change reflects in the results. In tournament 3, the planning agent achieved almost 60% of wins, which is a greater percentage than tournament 1, that happened in an environment with no destructible blocks. This higher percentage can be explained, because of the presence of destructible blocks, agents have limited ability to move, which makes it easier to trap an enemy. This fact is also the same reason the human player he wins more often and why the percentage of draws also decreases, when comparing the results of tournament 4 with the results of tournament 2.

The main conclusion of the experiments on this requirement is that the planning synthetic player is much more capable of playing and winning the game than any machine learning agent used in these experiments, but cannot reach the same performance of a human player.

4.5.3 Results of Requirement #2

The experiments on testing the ability of the synthetic players to play in new, unseen scenarios show the superiority of the planning synthetic player over all of the machine learning agents tested, similarly to the conclusions of requirement #1.

Fact worth noticing is that the planning agent actually wins even more in new scenarios, by obtaining more than 60% of wins in tournaments 1 and 3, compared to tournaments of requirement #1. Reason for this is that the machine learning agents are not trained in these new scenarios and therefore their performance is worse. This allows the planning synthetic player to win the game more easily.

The same pattern of performances for the requirement #1 applies to requirement #2. Despite the superiority of the planning synthetic player over the machine learning agents, it cannot deal with the human player effectively, obtaining a very small percentage of victories. The human player's skill remains unchanged whether he plays in scenarios with a new configuration of destructible blocks or not.

The machine learning agent of the 7th trial of 4th series is considerably better at playing the game than the one of the 6th trial when in an environment with 60% destructible blocks. Explanation to this lies in the type of environment the agents were trained in prior. The agent of the 6th trial was trained in environments with no destructible blocks, while the one of 7th was trained in environments with a percentage of 30 of that type of blocks. Therefore, it may be easier for the agent of the 7th to generalize its knowledge to a new environment configuration, since the other one is not used to dealing with that kind of game element in the environment.

Main conclusions with these experiments are: 1) the planning synthetic player presents a good generalization capability, being able to consistently beat the machine learning agents; 2) the planning synthetic player is not at the same level as the human player, with the latter dominating the former; 3) a machine learning agent that was trained in environments with the presence of destructible blocks can perform better in a new configuration than one that was trained in environments without such blocks.

4.5.4 Results of Requirement #3

To evaluate the credibility of the synthetic players, a research was conducted. Five people were asked to watch videos of the agents' gameplay. Their task was to classify the agents according to several attributes. The obtained results reveal that the planning agent is the most credible synthetic player, being rated 14 out of 20 (70%)times with the maximum score in the *Apparent Rational Competence* attribute. Furthermore, in the *Apparent Intentionality* attribute, the planning agent was rated 2, 12 out of 20 (60%)times, and the maximum rating of 3 on 4 occasions out of the total 20. Regarding the *Anticipatory Behaviour* attribute, the planning agent scored 2 half of the time (10 out of 20 times), and the maximum rating of 3, 6 out of 20 times (30%). The main reasons for these high scores, according to the survey participants, are:

- the agent has a clear notion of rules and is able to always move out of danger zones.
- it is able to plant bombs in order to destroy blocks.
- the agent's aggressiveness, i.e., it is constantly pursuing enemies trying to kill them.

The planning agent did not rate the maximum score in *Apparent Intentionality*, because according to the participants' shared interpretation, *"it tends to go away from the opponents to plant a bomb to destroy a random block"*. This may be caused by a possible fault in the agent's logic code, that makes it "think" that there is no goal of higher priority available than to destroy the nearest block, even when there is one.

The machine learning agent of the 6th trial of the 4th series of trials was the most credible one, but still obtained worse scores than the planning synthetic player. This particular machine learning agent presents a broad distribution of scores. For instance, in the *Apparent Rational Competence* attribute, it scored a 0, 9 out of 20 times, but also scored the maximum rating of 3 on 5 of the total 20 occasions. The other attributes are also evenly distributed in terms of score. This is due to the high inconsistency of the agent. It can both play the game efficiently and make non-reasonable actions that result in its early demise. This can be explained by the number of destructible blocks in the environment throughout a match. At the beginning, with the presence of this type of game element, that is unknown to the agent, it does not know what to do. However, if the agent manages to survive the early phase of the game, it will start to play better, because the longer the game takes, the less destructible blocks are left in the environment. Consequently, by playing better, its credibility also increases. In fact, the occasions when the agent got rated with maximum grade, in attributes like *Apparent Rational Competence* and *Apparent Anticipatory Behaviour*, are the ones referring to matches where the end-game is characterized by explosion of all blocks. This results in the environment being exactly the same like the one in which the machine learning synthetic player was trained. This also supports the comments of few survey participants, who state that most of the time, this agent *"roams the map, but knows how to avoid bombs"*, and *"in some occasions can put up a good challenge"* against the planning synthetic player.

The remaining machine learning agents did not produce relevant results, since they got a score of 0 in every attribute on the majority of occasions. This is because the early demise of the agents in most matches, killing themselves with their own bombs.

A random synthetic player was also used in some videos, to replace the human player. This random agent managed to score better than the machine learning agents of the 7th and the 9th trials of 4th series in two attributes: *Apparent Rational Competence* and *Apparent*

Intentionality. The participants stated that sometimes this agent *"looks like it knows how to avoid bombs"*, and can even *"plant a bomb when an enemy approaches"*. This is however, just the perception of the participants, since this synthetic player is random. Human brain being tricked into finding patterns and making sense of things, even when they do not, is the reason for this.

Finally, the human player scored the maximum at each attribute the majority of time, revealing by comparison that only the planning synthetic player can come a little close to such results.

This page is intentionally left blank.

Chapter 5

Conclusion and Future Work

5.1 Contributions

This dissertation's main goal was to explore the possibility of using synthetic players as tools to design, play and test procedurally generated game scenarios. With the conclusion of this work, the following contributions were made:

- A planning synthetic player that is able to play procedurally generated game scenarios with success. If the action space does not change from scenario to scenario, the planning agent is able to play all of them without any modification required. Despite not being tested, the planning agent could, in theory, also play a different game scenario involving new actions and goals, without requiring too much developer's work. This could be possible due to the ease of adding new actions and goals to the planning synthetic player's logic and game state representation. The planning synthetic player also achieved relevant results regarding its believability.
- A machine learning synthetic player that, despite not performing at the same level of the planning synthetic player, it could be perceived to have grasped how to play the game and managed to survive long-enough periods to explore the game space. Its performance increases the more the game scenario resembles the training environment the agent was trained on.
- An analysis and comparison between the performance of both on game scenarios involving PCG. Several experiments were made in order to determine if the synthetic players fulfill the three main requirements identified *à priori*. They include: a series of 100 matches against a human player in different environment configurations, tournaments of 10000 matches between all the synthetic players, and a survey on the actions of the agents while playing the game.

5.2 Future Work

Due to time restraints, there are a few possibilities that were not explored within this work, but nonetheless deserve to be mentioned because they point to some future studies to undertake.

- One of the main goals of developing synthetic players is for testing game scenarios. Therefore, the creation of an evaluation function for game scenarios should be investigated. This function could find a way to map directly the performance of the synthetic player testing the game scenario with the quality of the scenario itself.

- The machine learning agent developed for this work did not produce relevant results. One of the possible reasons for its modest performance might be that the chosen representation was not the most suitable for the learning process. Training a new machine learning agent with a new state representation could be beneficial.
- Due to the ease of adding new goals and actions to it, the planning synthetic player is suitable for other case studies. A possible future direction is to take this synthetic player's modularity to the limit, putting it in game scenarios very different from the one of this work and evaluate its performance.
- An utility-based synthetic player is another idea that deserves exploration. The machine learning agent's rewards could provide a good starting point in order to define some proper values for the utility function of the synthetic player.

5.3 Closing Remarks

This work allowed me to deepen my knowledge about the research area I am more passionate about: AI in Games. I got more familiarized with the state-of-art AI techniques used, not only in academia but also in the game industry.

One valuable lesson learned was the pros and cons of certain AI techniques. On one hand, there is classic AI, with well-established methods, like planning and ad-hoc behaviours (such as behaviour trees and finite-state machines). These methods provide a safety net for the developers that do not want to take risks during the implementation process, since they are easy to debug and edit, despite their increased authorial burden. On the other hand, there is machine learning, with its easy setup, that greatly decreases the developer's work. However, machine learning can be a risky choice of methods because of the developer's lack of control over the obtained results.

Another lesson learned was the necessity of adopting a work planning methodology, which is crucial to any large project's completion on time.

References

N. Barreto, L. Macedo, and L. Roque. MultiAgent System Architecture in Orphibs II State of the Art. 2014.

N. Barreto, R. Craveirinha, and L. Roque. Designing a creature believability scale for videogames. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 10507 LNCS:257–269, 2017. ISSN 16113349. doi: 10.1007/978-3-319-66715-7_28.

J. Bates. The Role of Emotion in Believable Agents. *Communications of the ACM*, 37(7): 122–125, 1994. doi: 10.1145/176789.176803.

A. Beij. Behaviour Trees: The Cornerstone of Modern Game AI. *Game AI North, Copenhagen*, 2017. URL `https://www.guerrilla-games.com/read/the-ai-of-horizon-zero-dawn`.

M. E. Bratman, D. J. Israel, and M. E. Pollack. RESOURCE-BOUNDED PRACTICAL REASONING. 4(4):349–355, 1988.

E. Camilleri, G. N. Yannakakis, and A. Dingli. Platformer level design for player believability. *IEEE Conference on Computatonal Intelligence and Games, CIG*, 0, 2016. ISSN 23254289. doi: 10.1109/CIG.2016.7860404.

J. Campbell. *The Hero's Journey.* 1991. ISBN 9780062501714.

M. Campbell, A. J. Hoane, and F. H. Hsu. Deep Blue. *Artificial Intelligence*, 134(1-2): 57–83, 2002. ISSN 00043702. doi: 10.1016/S0004-3702(01)00129-1.

B. Correia. Modeling and Generation of Playable Scenarios Based on Societies of Agents. *Master' Dissertation, Universidade de Coimbra, 30041-531 Coimbra*, 2021.

C. A. Cruz and J. A. R. Uresti. HRLBˆ2: A reinforcement learning based framework for believable bots. *Applied Sciences (Switzerland)*, 8(12), 2018. ISSN 20763417. doi: 10.3390/app8122453.

M. P. Eladhari and M. Sellers. Good moods : Outlook, affect and mood in dynemotion and the mind module. *ACM Future Play 2008 International Academic Conference on the Future of Game Design and Technology, Future Play: Research, Play, Share*, pages 1–8, 2008. doi: 10.1145/1496984.1496986.

R. L. Epstein, Joshua M. Axtell. *Growing Artificial Societies: social science from the bottom up.* 1994. ISBN 0262550253.

D. Ferrucci, E. Brown, J. Chu-Carroll, J. Fan, D. Gondek, A. A. Kalyanpur, A. Lally, J. W. Murdock, E. Nyberg, J. Prager, N. Schlaefer, and C. Welty. Building watson: An overview of the deepQA project. *AI Magazine*, 31(3):59–79, 2011. ISSN 07384602. doi: 10.1609/aimag.v31i3.2303.

R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189–208, 1971. ISSN 00043702. doi: 10.1016/0004-3702(71)90010-5.

I. França, A. Paes, and E. Clua. *Learning How to Play Bomberman with Deep Reinforcement and Imitation Learning*, pages 121–133. 11 2019. ISBN 978-3-030-34643-0. doi: 10.1007/978-3-030-34644-7_10.

P. Garcia-Sanchez, A. Tonda, G. Squillero, A. Mora, and J. J. Merelo. Evolutionary deckbuilding in hearthstone. *IEEE Conference on Computatonal Intelligence and Games, CIG*, 0(Section IV), 2016. ISSN 23254289. doi: 10.1109/CIG.2016.7860426.

B. Gorman, C. Thurau, C. Bauckhage, and M. Humphrys. Believability testing and Bayesian imitation in interactive computer games. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 4095 LNAI:655–666, 2006. ISSN 16113349. doi: 10.1007/11840541_54.

M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver. Rainbow: Combining improvements in deep reinforcement learning. *32nd AAAI Conference on Artificial Intelligence, AAAI 2018*, pages 3215–3222, 2018.

P. Hingston. A turing test for computer game bots. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(3):169–186, 2009. ISSN 1943068X. doi: 10.1109/TCIAIG.2009.2032534.

C. Holmgård, A. Liapis, J. Togelius, and G. N. Yannakakis. Generative Agents for Player Decision Modeling in Games. *Proc. FDG 2014*, 2014.

C. Holmgard, A. Liapis, J. Togelius, and G. N. Yannakakis. Evolving personas for player decision modeling. *IEEE Conference on Computatonal Intelligence and Games, CIG*, 2014. ISSN 23254289. doi: 10.1109/CIG.2014.6932911.

C. Holmgård, A. Liapis, J. Togelius, G. Yannakakis, C. Holmgård, A. Liapis, J. Togelius, G. Yannakakis, C. Holmgård, A. Liapis, J. Togelius, and G. N. Yannakakis. Personas versus Clones for Player Decision Modeling. 2016.

C. Holmgård, M. C. Green, A. Liapis, and J. Togelius. Automated playtesting with procedural personas through MCTs with evolved heuristics. *IEEE Transactions on Games*, 11(4):352–362, 2019. ISSN 24751510. doi: 10.1109/TG.2018.2808198.

Y. Jacquier. The Alchemy and Science of Machine Learning for Games. *Game Developers Conference*, 2019. URL `https://www.youtube.com/watch?v=Eim_0jCQW_g`.

H. Jiang, J. M. Vidal, and M. N. Huhns. EBDI: An architecture for emotional agents. *Proceedings of the International Conference on Autonomous Agents*, pages 38–40, 2007. doi: 10.1145/1329125.1329139.

G. W. Lecky-Thompson. *AI and Artificial Life in Video Games*. 2008. ISBN 1584505583.

R. Li. Applying Reinforcement Learning to Develop Game AI In NetEase Games. *Game Developers Conference*, 2020. URL `https://www.youtube.com/watch?v=gXilr5C9MZs`.

A. Liapis, C. Holmgård, G. N. Yannakakis, and J. Togelius. Procedural personas as critics for dungeon generation. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9028:331–343, 2015. ISSN 16113349. doi: 10.1007/978-3-319-16549-3_27.

M. Lopes. Bomberman as an artificial intelligence platform. *Master's Dissertation, Universidade do Porto*, 2016.

A. B. Loyall. Believable Agents: Building Interactive Personalities, 1997. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.62.2436{&}rep=rep1{&}type=pdf`.

T. Mahlmann, J. Togelius, and G. N. Yannakakis. Evolving card sets towards balancing dominion. *2012 IEEE Congress on Evolutionary Computation, CEC 2012*, 2012. doi: 10.1109/CEC.2012.6256441.

R. R. Mccrae and O. P. John. The five-factor model: issues and applications. *Journal of personality*, 60(2):175–532, 1992. ISSN 0022-3506. URL `http://www.ncbi.nlm.nih.gov/pubmed/1635040`.

V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing Atari with Deep Reinforcement Learning. pages 1–9, 2013. URL `http://arxiv.org/abs/1312.5602`.

V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015. ISSN 14764687. doi: 10.1038/nature14236.

M. Moravčík, M. Schmid, N. Burch, V. Lisý, D. Morrill, N. Bard, T. Davis, K. Waugh, M. Johanson, and M. Bowling. DeepStack: Expert-level artificial intelligence in heads-up no-limit poker. *Science*, 356(6337):508–513, 2017. ISSN 10959203. doi: 10.1126/science.aam6960.

A. Nareyek. Game AI is dead. Long live game AI! *IEEE Intelligent Systems*, 22(1):9–11, 2007. ISSN 15411672. doi: 10.1109/MIS.2007.10.

J. Niklaus, M. Alberti, V. Pondenkandath, R. Ingold, and M. Liwicki. Survey of Artificial Intelligence for Card Games and Its Application to the Swiss Game Jass. *Proceedings - 6th Swiss Conference on Data Science, SDS 2019*, pages 25–30, 2019. doi: 10.1109/SDS.2019.00-12.

J. Orkin. Applying Goal-Oriented Action Planning to Games. *AI Game Programming Wisdom 2*, pages 217–227, 2003. URL `http://alumni.media.mit.edu/{~}jorkin/GOAP{_}draft{_}AIWisdom2{_}2003.pdf`.

J. Orkin. Three states and a plan: the AI of FEAR. *Game Developers Conference*, 2006(1):1–18, 2006. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.92.8551{&}rep=rep1{&}type=pdf{%}5Cnhttp://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.92.8551{&}rep=rep1{&}type=pdf`.

J. Ortega, N. Shaker, J. Togelius, and G. N. Yannakakis. Imitating human playing styles in Super Mario Bros. *Entertainment Computing*, 4(2):93–104, 2013. doi: 10.1016/j.entcom.2012.10.001.

J. Pfau, A. Liapis, G. Volkmar, G. N. Yannakakis, and R. Malaka. Dungeons Replicants: Automated Game Balancing via Deep Player Behavior Modeling. *IEEE Conference on Computatonal Intelligence and Games, CIG*, 2020-Augus:431–438, 2020. ISSN 23254289. doi: 10.1109/CoG47356.2020.9231958.

J. Rasmussen. Are Behavior Trees a Thing ofthe Past? *Gamasutra*, 2016. URL https://www.gamasutra.com/blogs/JakobRasmussen/20160427/271188/Are_Behavior_Trees_a_Thing_of_the_Past.php.

S. Russell and P. Norvig. *Artificial Intelligence - A Modern Approach*, volume 4. 2010. ISBN 9781424455850. doi: 10.1109/ICCAE.2010.5451578.

J. Ryan. Curating Simulated Storyworlds. (March):816, 2018. URL https://escholarship.org/uc/item/1340j5h2.

J. O. Ryan, A. Summerville, M. Mateas, and N. Wardrip-fruin. Toward Characters Who Observe, Tell, Misremember, and Lie. *Proceedings of the 2nd Workshop on Experimental AI in Games*, (November 2015):56–62, 2015. URL https://users.soe.ucsc.edu/{~}jor/publications/ryanTowardCharactersWhoObserveTellMisrememberLie.pdf.

J. O. Ryan, B. Samuel, and A. J. Summerville. Bad news: A game of death and communication. *Conference on Human Factors in Computing Systems - Proceedings*, 07-12-May-(May 2016):160–163, 2016. doi: 10.1145/2851581.2890375.

T. Schaul, J. Togelius, and J. Schmidhuber. Measuring Intelligence through Games. (2007):1–19, 2011. URL http://arxiv.org/abs/1109.1314.

N. Shaker, J. Togelius, G. N. Yannakakis, L. Poovanna, V. S. Ethiraj, S. J. Johansson, R. G. Reynolds, L. K. Heether, T. Schumann, and M. Gallagher. The turing test track of the 2012 Mario AI Championship: Entries and evaluation. *IEEE Conference on Computatonal Intelligence and Games, CIG*, 2013. ISSN 23254270. doi: 10.1109/CIG.2013.6633634.

J. Shih and E. Teng. Successfully Use Deep Reinforcement Learning In Testing and NPC Development, 2020. URL https://www.youtube.com/watch?v=Q5RAE73zCKQ.

D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016. ISSN 14764687. doi: 10.1038/nature16961. URL http://dx.doi.org/10.1038/nature16961.

D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144, 2018. ISSN 10959203. doi: 10.1126/science.aar6404.

P. Spronck, I. Sprinkhuizen-Kuyper, and E. Postma. Online Adaptation of Game Opponent AI with Dynamic Scripting. *Proceedings of the 5th International Conference on Intelligent Games and Simulation (GAME-ON 2004)*, pages 33–37, 2004.

T. Thompson. The Campaign AI of Total War: Rome II (Part 3 of 5), 2018. URL https://www.youtube.com/watch?v=1m9-7ZrpbBo.

T. Thompson. The AI of Half-Life: Finite State Machines, 2019a. URL https://www.youtube.com/watch?v=JyF0oyarz4U.

T. Thompson. Behaviour Trees: The Cornerstone of Modern Game AI, 2019b. URL https://www.youtube.com/watch?v=6VBCXvfNlCM.

T. Thompson. Building the AI of F.E.A.R. with Goal Oriented Action Planning, 2020. URL `https://www.youtube.com/watch?v=PaOLBOuyswI`.

J. Togelius, R. De Nardi, and S. M. Lucas. Towards automatic personalised content creation for racing games. *Proceedings of the 2007 IEEE Symposium on Computational Intelligence and Games, CIG 2007*, pages 252–259, 2007. doi: 10.1109/CIG.2007.368106.

J. Togelius, G. N. Yannakakis, S. Karakovskiy, and N. Shaker. Assessing believability. *Believable Bots: Can Computers Play Like People?*, 9783642323:215–230, 2012. doi: 10.1007/978-3-642-32323-2_9.

V. Vaishnavi, W. Kuechler, and S. Petter. Design Science Research in Information Systems. 2004. URL `http://www.desrist.org/design-research-in-information-systems/`.

O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, J. Oh, D. Horgan, M. Kroiss, I. Danihelka, A. Huang, L. Sifre, T. Cai, J. P. Agapiou, M. Jaderberg, A. S. Vezhnevets, R. Leblond, T. Pohlen, V. Dalibard, D. Budden, Y. Sulsky, J. Molloy, T. L. Paine, C. Gulcehre, Z. Wang, T. Pfaff, Y. Wu, R. Ring, D. Yogatama, D. Wünsch, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, K. Kavukcuoglu, D. Hassabis, C. Apps, and D. Silver. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019. ISSN 14764687. doi: 10.1038/s41586-019-1724-z. URL `http://dx.doi.org/10.1038/s41586-019-1724-z`.

M. Wooldridge. An introduction to multi agent systems. *Technical Paper - Society of Manufacturing Engineers*, TP05PUB1, 2005.

G. N. Yannakakis. Game AI revisited. *CF '12 - Proceedings of the ACM Computing Frontiers Conference*, pages 285–292, 2012. doi: 10.1145/2212908.2212954.

G. N. Yannakakis and J. Togelius. A Panorama of Artificial and Computational Intelligence in Games. *IEEE Transactions on Computational Intelligence and AI in Games*, 7(4): 317–335, 2015. ISSN 1943068X. doi: 10.1109/TCIAIG.2014.2339221.

G. N. Yannakakis and J. Togelius. Artificial intelligence and games. *Artificial Intelligence and Games*, pages 1–337, 2018. doi: 10.1007/978-3-319-63519-4.

Y. Zhao, I. Borovikov, F. De Mesentier Silva, A. Beirami, J. Rupert, C. Somers, J. Harder, J. Kolen, J. Pinto, R. Pourabolghasem, J. Pestrak, H. Chaput, M. Sardari, L. Lin, S. Narravula, N. Aghdaie, and K. Zaman. Winning Is Not Everything: Enhancing Game Development with Intelligent Agents. *IEEE Transactions on Games*, 12(2):199–212, 2020. ISSN 24751510. doi: 10.1109/TG.2020.2990865.

M. Zinkevich, M. Johanson, M. Bowling, and C. Piccione. Regret minimization in games with incomplete information. *Advances in Neural Information Processing Systems 20 - Proceedings of the 2007 Conference*, pages 1–8, 2009.

Appendices

This page is intentionally left blank.

Appendix A

An example of an action script:

```
//EXTRANEOUS DETAIL OMITTED

public class ActionMoveUp : SymbolicAction
{
    //Initializes the attributes of the action
    public override void Init(PlanningSyntheticPlayer agent)
    {
        Agent = agent;
        RawAction = (int)Action.MoveUp;
        Effect = SyntheticPlayerUtils.deepCopyWorld(agent.GridArray);
    }

    //Reverts the effects of the action.
    //It is like it never happened
    public override void Revert()
    {
        Agent.SimulatedY -= 1;

    }

    //Simulates the action in the environment, applying its effects
    public override void Simulate()
    {
        if (Effect[Agent.SimulatedX, Agent.SimulatedY] ==
                                    (int)Tile.PlayerNBomb)
        {
            Effect[Agent.SimulatedX, Agent.SimulatedY] =
                                            (int)Tile.Bomb;
        }
        else if (Effect[Agent.SimulatedX, Agent.SimulatedY] ==
                                            (int)Tile.Player)
        {
            Effect[Agent.SimulatedX, Agent.SimulatedY] =
                                        (int)Tile.Walkable;
        }
        if (Agent.SimulatedY + 1 < Agent.GridArray.GetLength(1))
        {
            if (Effect[Agent.SimulatedX, Agent.SimulatedY + 1] ==
                                            (int)Tile.Walkable)
            {
                Effect[Agent.SimulatedX, Agent.SimulatedY + 1] =
                                                (int)Tile.Player;
                Agent.SimulatedY += 1;
            }
            else if (Effect[Agent.SimulatedX, Agent.SimulatedY + 1]
                                            == (int)Tile.Fire)
            {
                Effect[Agent.SimulatedX, Agent.SimulatedY + 1] =
(int)Tile.FireNPlayer;
                Agent.SimulatedY += 1;
            }
            else if (Effect[Agent.SimulatedX, Agent.SimulatedY + 1]
                                            == (int)Tile.Bomb)
            {
                Effect[Agent.SimulatedX, Agent.SimulatedY + 1] =
                                        (int)Tile.PlayerNBomb;
                Agent.SimulatedY += 1;
            }
        }
    }

    //Checks if the action is possible to be simulated
    public override bool CheckPreconditions(int [,] grid)
    {
        if (grid[Agent.SimulatedX, Agent.SimulatedY] ==
                                    (int)Tile.PlayerNBomb)
        {
            return false;
        }

        if (Agent.SimulatedX == Agent.position.x &&
            Agent.SimulatedY + 1 == Agent.position.y)
        {
            return true;
        }

        return SyntheticPlayerUtils.IsTileWalkableSim(grid,
                    Agent.SimulatedX, Agent.SimulatedY + 1);
    }

    //Checks if the action is possible to be executed
    //in the current game state
    public override bool IsPossible(int [,] grid)
    {
        if (SyntheticPlayerUtils.IsTileWalkable(grid,
            Agent.position.x, Agent.position.y + 1))
        {
            if (!SyntheticPlayerUtils.IsTileSafe(grid,
                new int[2]{ Agent.position.x, Agent.position.y})
                || SyntheticPlayerUtils.IsTileSafe(grid,
                new int[2]{Agent.position.x, Agent.position.y + 1}))
            {
                return true;
            }
        }
        return false;
    }
}
```

This page is intentionally left blank.

Appendix B

An example of a goal script:

```
public class BeSafeGoal : Goal
{
  /* Returns the grid that represents the goal game state
   * int[,] currentGrid: grid that represents
   *                         the current game state
   * int index: Index of the Goal Tile
   *           (tile where the agent wants to be)
   * PlanningSyntheticPlayer agent: reference to the planning agent
   */
   public override int[,] GetGoalGrid(int[,] currentGrid,
                  int index, PlanningSyntheticPlayer agent)
   {
       int[,] goalGrid =
           SyntheticPlayerUtils.deepCopyWorld(currentGrid);
       int[] goalTile =
               SyntheticPlayerUtils.GetTileFromIndex(index,
                                 currentGrid.GetLength(0));
       goalGrid[goalTile[0], goalTile[1]] = (int)Tile.Player;
       if (goalGrid[agent.position.x, agent.position.y] ==
                                           (int)Tile.Player)
       {
           goalGrid[agent.position.x, agent.position.y] =
                                       (int)Tile.Walkable;
       }
       else if (goalGrid[agent.position.x, agent.position.y] ==
                                       (int)Tile.PlayerNBomb)
       {
           goalGrid[agent.position.x, agent.position.y] =
                                           (int)Tile.Bomb;
       }
       agent.SimulatedX = goalTile[0];
       agent.SimulatedY = goalTile[1];
       return goalGrid;

   }

  /*Provides a Custom Heuristic to be used in A*.
   *This particular case is the Manhattan Distance.
   * WorldNode state: Current Node
   * WorldNode goal: Goal Node
   */
   public override double Heuristic(WorldNode state, WorldNode goal)
   {
       int[] start = new int[2]{
               state.Agent.SimulatedX, goal.Agent.SimulatedY};
       int[] end = new int[2] {
               goal.Agent.position.x, goal.Agent.position.y };
       return SyntheticPlayerUtils.CalculateManhattanDistance(
                                                        start, end);
    }

    /*
     * Checks if a particular node is the goal node
     * WorldNode node: Node to be checked
     */
    public override bool IsObjective(WorldNode node)
    {
        return node.Agent.SimulatedX == node.Agent.position.x &&
                node.Agent.SimulatedY == node.Agent.position.y;
    }

    //Checks if the Goal is Possible to Execute
    public override bool IsPossible()
    {
        RefTile = new int[2]{
            PlanningAgent.position.x, PlanningAgent.position.y};

        if (!SyntheticPlayerUtils.IsTileSafe(
                            PlanningAgent.GridArray, RefTile))
        {
            TargetTiles =
            SyntheticPlayerUtils.dangerTiles(
                                SyntheticPlayerUtils.dangerMap(
                                PlanningAgent.GridArray), true);
            return true;
        }
        return false;
    }
}
```