Implementation of bots for multi-action adversarial game solving

Diego Villabrille Seca

Final Degree Work
Bachelor's Degree in
Video Game Design and Development
Universitat Jaume I

June 30, 2022

Supervised by: Raúl Montoliu Colás

To Darius, who helped me stay sane when I was struggling and was always there for me.

To Javier, who keeps my interest in programming alive by always having a new topic to discuss.

To my mom, who endured endless days of listening to my nonsense with infinite patience and interest.

Acknowledgments

First of all, I would like to thank my Final Degree Work supervisor, Raúl Montoliu, for his praiseworthy effort to keep me on track with this project, for making my interest in researching artificial intelligence grow with each meeting and for encouraging me to publish my work every time I get an opportunity.

I want to thank Marta Martín Núñez too, for being the fist professor who got me into writing an academic paper and for being such a good role model and counselor for me since my first year in university, sparking my interest in academia.

I also want to thank the GIANT-UJI and ITACA-UJI research groups for being my two homes within the university. They allowed me to have a multidisciplinary approach to video games that you can only acquire by getting perspectives from their very different research topics, and they provided me with all the resources and training I could have ever hoped for.

I would finally like to thank Sergio Barrachina Mir and José Vte. Martí Avilés for their really useful LaTeX template for writing the Final Degree Work report, which has served as a starting point for this document.

Abstract

This document presents the report of a Final Degree Work consisting of a project that includes several components related to multi-action adversarial game solving. It involves the development of a multi-action adversarial card game that is capable of delivering relevant results on the performance of different algorithms. It also encompasses the development and testing of bots that play the game using several different algorithms, one of which entails a novel approach to solving these kind of games. Finally, as the main research action of the project, an experiment on the performance of the algorithms within the game was performed.

Within each of the chapters of this report attention will be put into going trough all these components of the project. The framework architecture for A Simple Multi-Action Card Game (ASMACAG) will be firstly presented. It is the simple but complete card game proposed as a tool to test, develop and debug bots that implement artificial intelligence algorithms in the context of adversarial multi-action games. Then the focus will be on the development process and the specifics of each of the algorithms included in the project, as well as the decisions taken on what parameters to use for them. Finally the experiment carried out within the game will be discussed, to further comprehend what conclusions can be drawn about the implemented bots and the game itself.

Contents

List of Figures

List of Tables

CHAPTER 1

INTRODUCTION

Contents

This chapter present the project carried on. It goes trough the different components it includes, stating the objectives to meed and the way the idea was conceived.

1.1 Work Motivation

This project stems from the analysis of the latest developments within the academic discipline of artificial intelligence and the possible applications on video games it can have [16]. Even though artificial intelligence techniques are established in academia, they are not systematically applied industry-wide yet. But it is clear that there is a push to use some of them for game AI, specially machine learning techniques, and they have already been successfully applied in several other industries. It is in this context that the need for a project that specialises in the emergent field of artificial intelligence applied to video games is born. The scope of this work is specifically constrained to multi-action adversarial games, since they present a problem complex enough to yield useful results and can resemble several existing games within the current video game industry.

The development of a specific game to test the different algorithms is a need within a field populated by increasingly complex tools for playing games that either do not meet the requirements for testing all kinds of algorithms or aren't designed to meet a good enough usability standard. This need is the origin of A Simple Multi-Action Card Game

(ASMACAG). It is required that it provides a simple enough context in which the bots can be tested and understood and that it allows for easy step by step debugging, while being complex enough that the results it yields are statistically relevant.

The other motivation behind this whole project is to allow a deeper understanding of some algorithms and their behavior. There is a drive to test how they perform and what actions they decide on, both when paired with each other and when paired with a user interface that allows a human player to interact with the bots. For these reasons, the algorithms to be tested have been selected based on several different motivations. Some of them are the most researched and currently relevant algorithms that can be applied to multi-action adversarial video game solving. These are the Monte Carlo Tree Search Algorithm [1] and the Online Evolution Algorithm [6]. One of them, the N-Tuple Bandit Evolutionary Algorithm [9], has been selected with the aim of innovating within the field, by using an evolutionary algorithm originally designed for parameter optimization that to the best of this researcher's knowledge has not been applied yet to live game solving. The other algorithms, which are Random and Greedy One-Step Lookahead, were picked to act as a baseline benchmark for performance.

1.2 Objectives

There are three main objectives within the project. The fist one is the implementation of the final version of ASMACAG, for which the original idea and first concept was presented in [13]. Within this tool's design and development the main goals to accomplish a satisfactory result are simplicity, efficacy, cleanliness and efficiency. Focusing on these goals should yield a tool that is extremely useful for researching, studying and teaching artificial intelligence algorithms.

The second target of the project is to implementing bots that play using different algorithms. These must be picked according to the criteria described in *Section 1.1* and tested to make sure their behaviour and performance is adequate.

The last objective of this work is to analyse the performance of the different algorithms against each other in the game. It is of special interest to study the results yiended by the N-Tuple Bandit Evolutionary Algorithm, since this is the most innovative approach of the ones presented. As part of this process there is also an intention to evaluate ASMACAG itself as a tool, so that the level of completion of the aforementioned goals can be assesed.

1.3 Environment and Initial State

This work started to be shaped within the context of my scholarships for collaboration with the Research Group on Machine Learning for Smart Environments (GIANT-UJI). It was there where my interest for learning about AI algorithms sparked. Then I started to research about how AI, and more specifically machine learning, is applied to solving video games.

I started to experiment with multi-action adversarial games by using *Hero Academy*, which is presented in [6] along the Online Evolution Algorithm. At the same time I also learned about the most commonplace algorithms and techniques for this by researching and reading papers. Within this process I was presented with the concept of the N-Tuple Bandit Evolutionary Algorithm, as defined in [9], and with the idea of using it to optimise a set of game actions instead of a set of parameters, in the same way that the Online Evolution Algorithm or the Rolling Horizon Algorithm [12] do. It is in this context and given that combination of specific needs that the seed and the idea for ASMACAG and for this work itself was developed.

CHAPTER 2

Planning and resources evaluation

Contents

This chapter deals with the technical planing of the project, addressing the timeline and the material and human resources needed. It also includes an estimation of the economic cost of those resources.

2.1 Planning

In this section, the detailed time planning of the work is presented as a list of tasks. The time devoted to each of the tasks is detailed, as well as the task's description and whether it has changed from the moment it was originally planed. Using an estimated dedication of 14 hours per week the timeline of the project is clarified, with the task dependencies being showcased in a Gantt chart at the end of the section. The tasks needed for this work's completion are as follows.

1. **Planning and initial proposal (15 hours):** This task consists of envisioning the basics of the project concept and developing the first version of the planning presented in here.

2. **Research, analysis and design (40 hours):** This task includes the research time needed to understand the algorithms that will be used during the project as well as to envision and give shape to the architecture of ASMACAG as a tool. It also includes the development of an analysis and design document that summarises

all aspects of the project design and architecture. That document is further developed during the documentation task, into what has become *Chapter 3* of this report.

3. **Game implementation (70 hours):** The implementation of ASMACAG itself is included in this task. This encompasses the programming work for creating it as well as the process of debugging it, while following the guidelines established when designing the game framework. It also includes generating documentation for the code that allows other programmers to use it as a tool for various kinds of projects, as per the process also envisioned during the previous task.

4. **Basic algorithms implementation (12 hours):** This task consists of developing some basic algorithms as bots for the game, specifically Random Algorithm and One Step Lookahead Algorithm. This serves as a baseline for performance testing and debugging of other bots and can help identify any error within the game framework early.

5. **Human Player implementation (13 hours):** This task is composed of the development of bot that allows a human player to play ASMACAG using the command line as a simple user interface for interaction. This, once again, is useful for debugging other bots and can help identify any error within the game framework early. Originally this task was envisioned to be longer and include a GUI for the human player, but this idea was discarded during the design process as explained and justified in *Subsection 3.5.2*.

6. **MCTS algorithm implementation (20 hours):** This task encompasses the implementation of the Monte Carlo Tree Search Algorithm into a bot that can play ASMACAG. It also includes the debugging process, which is assisted by the algorithms implemented previously and the tools provided by the framework itself.

7. **OE algorithm implementation (30 hours):** The implementation of the Online Evolution Algorithm into a bot that can play ASMACAG is the main part of this task. It includes the debugging process too, which is, once again, assisted by the algorithms implemented previously and the tools provided by the framework itself.

8. **NTBEA implementation (40 hours):** This task consists of the development of a bot that plays ASMACAG by applying the N-Tuple Bandit Evolutionary Algorithm. As the previous tasks it also includes the debugging process, assisted by the basic algorithms already implemented and the tools provided by the ASMACAG framework.

9. **Experiments and analysis (45 hours):** For this task several experiments are performed. Firstly, for each of the algorithms that take parameters to define their behaviour one experiment to decide on what parameters to use is included. Finally, a complete experiment to assess the performance of all the developed algorithms against each other is needed to complete the project.

10. **Report and documentation (45 hours):** This last task consists of elaborating this report document and all the additional materials included within it.

To conclude and clarify the development plan, a summary of this planning information, including the time frame and dependencies for each of the tasks, is conveyed in the Gantt chart at *Figure 2.1*, shown bellow.

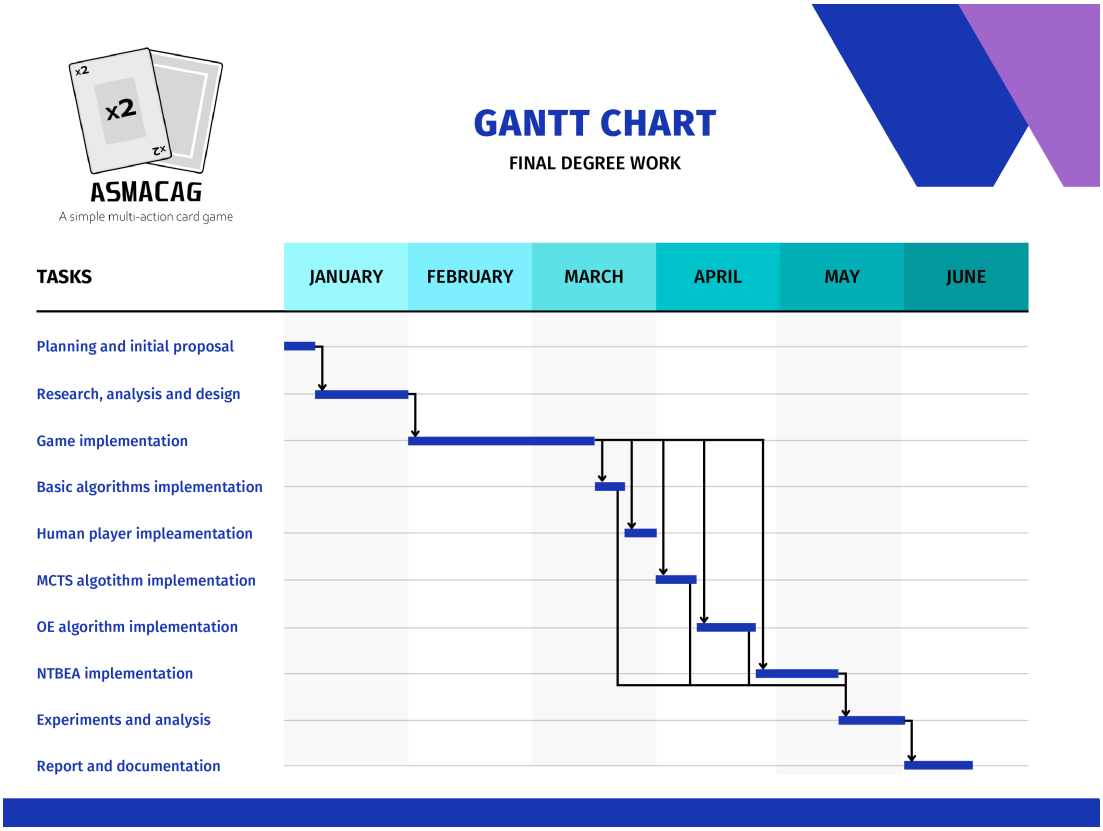


Figure 2.1: Estimated development timeline and dependencies between tasks as a Gantt chart.

Finally, the real development time for each of the tasks is provided, so that it can be compared with the original estimation. The justification for any discrepancies with the estimation is also laid out. This information is summarised at the Gantt chart shown in *Figure 2.2*. In total the project took about 20 hours over the initial estimation, which is within a reasonable margin of error.

1. **Planning and initial proposal:** 15 hours. In line with estimation.
2. **Research, analysis and design:** 40 hours. In line with estimation.
3. **Game implementation:** 60 hours. Development time was a moderately faster than estimated (10 hours less), given that the architecture was already thoroughly thought of during the previous step.

4. **Basic algorithms implementation:** 12 hours. In line with estimation.
5. **Human Player implementation:** 13 hours. In line with estimation.
6. **MCTS algorithm implementation:** 20 hours. In line with estimation.
7. **OE algorithm implementation:** 40 hours. 10 hours over the original estimated plan, given that some extra debugging was needed to polish bot behaviour.
8. **NTBEA implementation:** 55 hours. 15 hours over the original estimated plan. There were several bugs making the bot perform poorly that needed to be investigated and separately addressed, causing a delay during development.
9. **Experiments and analysis:** 45 hours. In line with estimation.
10. **Report and documentation:** 50 hours. Moderately over the estimation (5 extra hours).

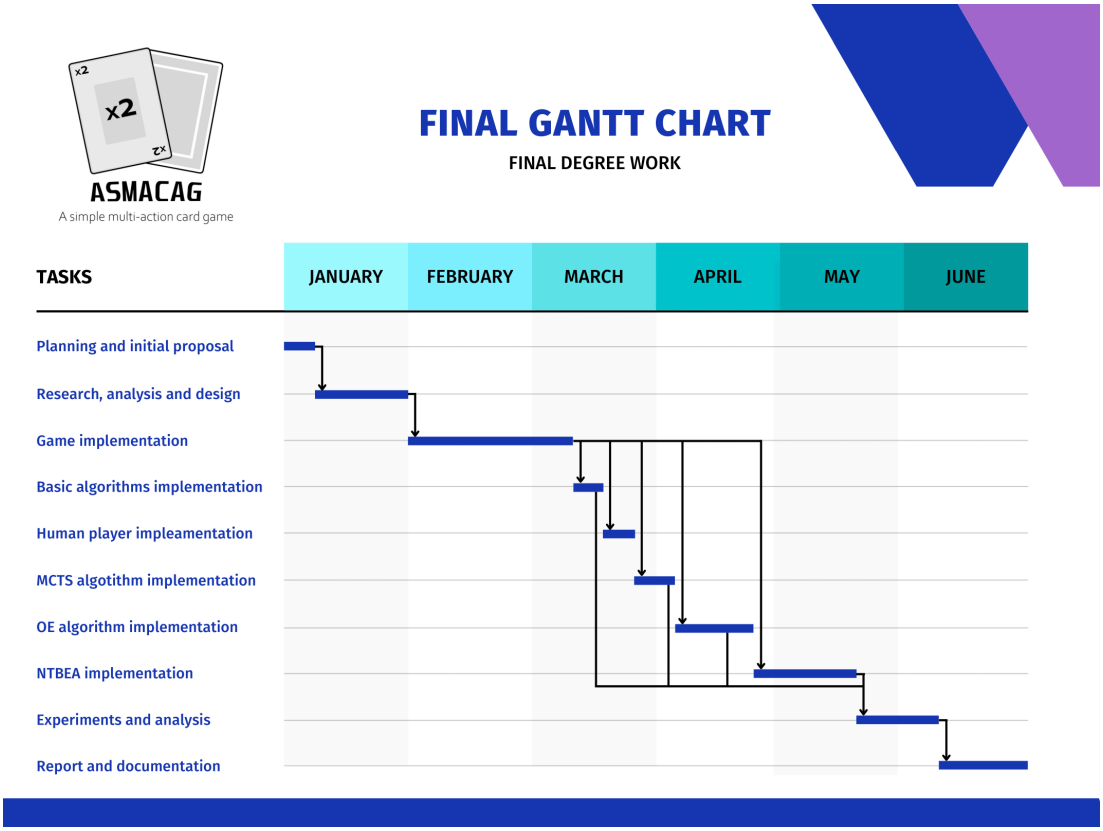


Figure 2.2: Real development timeline and dependencies between tasks as a Gantt chart.

2.2 Resource Evaluation

The costs of the work undertaken must be estimated in advance, therefore this section includes that estimation, focusing on both the human and the technical resources needed.

The human resources for this work consist of a total of 330 hours of project development as described in *Section 2.1*. The economic cost of hiring a junior software engineer for this amount of time would be of 3 500€. This has been calculated taking into account that the average hourly salary for a junior programmer in Spain is of 10.41€, as stated in [5].

Regarding hardware technical resources, the specs of the computer where the code was run during development and experimentation are the following. Note that the GPU is not included, given that ASMACAG does not make use of its resources.

- **Processor:** Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz. Market value of 150€.
- **RAM:** 16 GB DDR3. Market value of 60€.
- **Storage:** 500 GB SSD. Market value of 90€.

Also regarding technical resources, several software utilities are used within this project. This is a comprehensive list of all of them, the use they have within the project and their cost, if any.

- ***Ubuntu 20.04.4 LTS:*** Operating system for the computer described above.
- ***Python 3.9.7:*** The programming language used for the whole development process.
- ***PyCharm:*** IDE for Python with an extensive tool set for debugging the project. Cost of 25€ per month, adding up to 150€.
- ***Github Copilot:*** Tool for AI assisted programming that enables a quicker coding process, use described in more detail in *Subsection 4.1.1*.
- ***Git:*** Version control system used during the project for the source code.
- ***Fork:*** GUI for using *Git* that makes version control quicker and easier. Cost of 50€.
- ***Github:*** *Git* repository hosting service used to host the source code.
- ***Github Pages:*** Static website hosting service for associated *Github* repositories, used to host a live documentation for the source code.
- ***pdoc3:*** Python package and tool used for generating a complete documentation for ASMACAG and the bots by extracting it from the docstrings and type hints in the code.

- ***Overleaf:*** Online tool for LaTeX document editing, used for generating this report.
- ***Canva:*** Online tool for image and document editing, used for generating some of the figures used within the report.
- ***Lucidchart:*** Online tool for diagram design, used for generating some of the figures used within the report.
- ***Google Sheets:*** Online tool for spreadsheet editing, used for quickly processing some of the output data from the game and to generate the charts available as figures in this report.

Taking all the above technical resources and the human resources into account the total economic cost of the project execution is **4 000€**.

CHAPTER 3

System Analysis and Design

Contents

This document section constitutes the technical foundations of the project. It displays the design of A Simple Multi-Action Card Game (ASMACAG) as a framework from a software engineering standpoint and the considerations and practises to follow during the development process.

ASMACAG's framework architecture will be firstly presented. It is a simple card game proposed as a tool to test, develop and debug bots that implement artificial intelligence algorithms in the context of multi-action games. This report goes through its rules first to then deduce the architecture that stems from them, taking into consideration some software engineering best practices. Some attention will be also given to user interface and user experience. Finally, the design of the experiments to be carried out within the game will be described, as well as the experimental process that comes along it.

3.1 Game Rules

The rules for ASMACAG are simple by design, since the game is thought of as a tool whose aim is to be as straightforward as possible. This design philosophy allows to easily step-by-step debug the bots and understand their decisions while also staying complex enough for the results to be relevant and representative of their performance. The rules in the base version are as follows. Note that each player of the game is a bot following an specific algorithm for decision-making.

- It is an adversarial game with 2 players, each of them is dealt 9 cards.
- There is a board with 20 cards dealt on it, which must contain only numbers.
- The cards include numbers from *1* to *6* and the special values *x2* and *%2*. The deck where the card are dealt from contains 8 cards of each of the numbers from *2* to *5*, 5 instances of *1* and *6* cards and 6 of each one of the special cards.
- Each player must play 3 cards per turn with no possibility to skip. Playing a card consists of taking it from your hand and placing it on any card available on the board, if it is numbered, or just using it directly if it has an special value. After playing a card both the played card and the used card from the board, if applicable, will be discarded. Turns alternate until there are no cards left on either player's hand.
- Each time a numbered card is played $(P - B) * F$ points are awarded to the player that used it, where P is the value of the card played, B is the value of the card on the board and F is a factor that defaults to 1. Special cards modify the value of F when computing the score of the next numbered card played by any of the players. The *x2* special card will duplicate F while the *%2* card will halve it.

An aspect taken into account in the design of the ASMACAG software is the need for flexibility in the test environment, specially regarding the rules. All these rules need to be programmed to be easily modified as needed during the testing process, by just coding a class that implements a forward model compatible with the game. Also, the parameters of the rules need to be easily accessible and changeable. This parameters include things, such as the number of cards dealt at each time, the amount of existing cards of each type, the range of numbers in the numbered cards, the number of actions to play per turn, etc.

This approach should allow not only for flexibility within this project during its development but also for ASMACAG to be useful as a tool for activities ranging from research to education, given that it allows for the implementation of almost any multi-action adversarial card game that uses a similar set of cards. Essentially the aim is to decouple the game from its rules and parameters up to a point where a new game can be implemented by just developing a forward model and a set of parameters.

3.2 Requirement Analysis

This section describes both the functional and the non-functional requirements that stem from the game rules for ASMACAG that have been presented above. Its objective is allowing to clarify the actual requirements of the developed framework, so that they are adequately fulfilled when designing the architecture and developing the tool.

3.2.1 Functional Requirements

- **R01:** The user can start a game between the bots.
- **R02:** The bots can get the current status of the game.
- **R03:** The bots can play special value cards.
- **R04:** The bots can play numbered cards.

3.2.2 Non-functional Requirements

- **R05:** The board has 20 cards.
- **R06:** The numbered cards values go from from *1* to *6*.
- **R07:** The special value cards are *x2* and *%2*.
- **R08:** There are 8 cards with each of the numbers from *2* to *5* in the deck, 5 cards with numbers *1* and *6* and 6 cards with each of the special values.
- **R09:** The numbered cards played on the board are worth $(P - B) * F$ points.
- **R10:** Each turn 3 cards must be played.

3.3 System Design

This section presents the architectural design of the ASMACAG framework trough the use descriptive information, use case tables and some UML diagrams. It provides a deeper understanding of the approach taken for developing and the design philosophy used.

3.3.1 Use cases

The different use cases for the game that stem from the requirements in *Section 3.2* are presented below, in *Table 3.1*, *Table 3.2*, *Table 3.3* and *Table 3.4*.

Table 3.1
Use case «UC1: Start game»

Use case	UC1: Start game
Requirement	The user can start a game between the bots.
Actors	User and both bots.
Description	A game can be started by the user by running the corresponding file with the needed settings.
Preconditions:	1. The bots to play must be indicated in the file.
Basic path:	1. User runs the file. 2. Game is played. 3. Results are yielded as a text file or as console output.
Alternative paths:	[2.1] An error happens running the game. [2.2] The operation is aborted yielding an error message as console output.

Table 3.2
Use case «UC2: Get game status»

Use case	UC2: Get game status
Requirement	R02: The bots can get the current status of the game.
Actors	Bot making the request.
Description	A bot can get the game status on its turn and get back all the information available to it as a player (board cards, hand cards and an unordered collection of the rest of cards left).
Preconditions:	1. The bot must be on its turn.
Basic path:	1. Bot requests the information. 2. The system creates a copy of the current game state. 3. The framework obfuscates the information that shouldn't be available to the requesting bot on the copy. 4. The framework returns the copy to the bot.
Alternative paths:	None.

Table 3.3
Use case «UC3: Play special card»

Use case	UC3: Play special card
Requirement	R03: The bots can play special value cards.
Actors	Bot playing the card.
Description	A bot can play a special value card, modifying the value of F.
Preconditions:	1. The bot must be on its turn.
Basic path:	1. Bot plays the card. 2. The framework modifies the stored value of F. 3. The framework removes the played card from the bot's hand.
Alternative paths:	[1.1] The bot has made an invalid movement, like playing a card not in its hand. [1.2] The framework gives the bot a big amount of negative points to discourage this action. [3a.1] The bot has played its third card. [3a.2] The framework passes the turn to the next player. [3b.1] There are no cards anymore on either player's hand or there are no cards anymore on the table. [3b.2] The game ends yielding the corresponding result.

Table 3.4
Use case «UC4: Play numbered card»

Use case	UC4: Play numbered card
Requirement	R04: The bots can play numbered cards.
Actors	Bot playing the card.
Description	A bot can play a numbered card from its hand on a numbered card on the board and get points from it.
Preconditions:	1. The bot must be on its turn.
Basic path:	1. Bot plays the card over a card on the board. 2. The framework adds $(P - B) * F$ points to the score of that bot. 3. The framework removes the played card from the bot's hand and the used card from the board. 4. The framework resets F to 1.
Alternative paths:	[1.1] The bot has made an invalid movement, like playing a card not in its hand or playing it over a card that is not on the board. [1.2] The framework gives the bot a big amount of negative points to discourage this action. [3a.1] The bot has played its third card. [3a.2] The framework passes the turn to the next player. [3b.1] There are no cards anymore on either player's hand or there are no cards anymore on the table. [3b.2] The game ends yielding the corresponding result.

3.3.2 Use case diagram

The UML use case diagram in *Figure 3.1* describes how the use cases presented before interact between them and with the actors of the system.

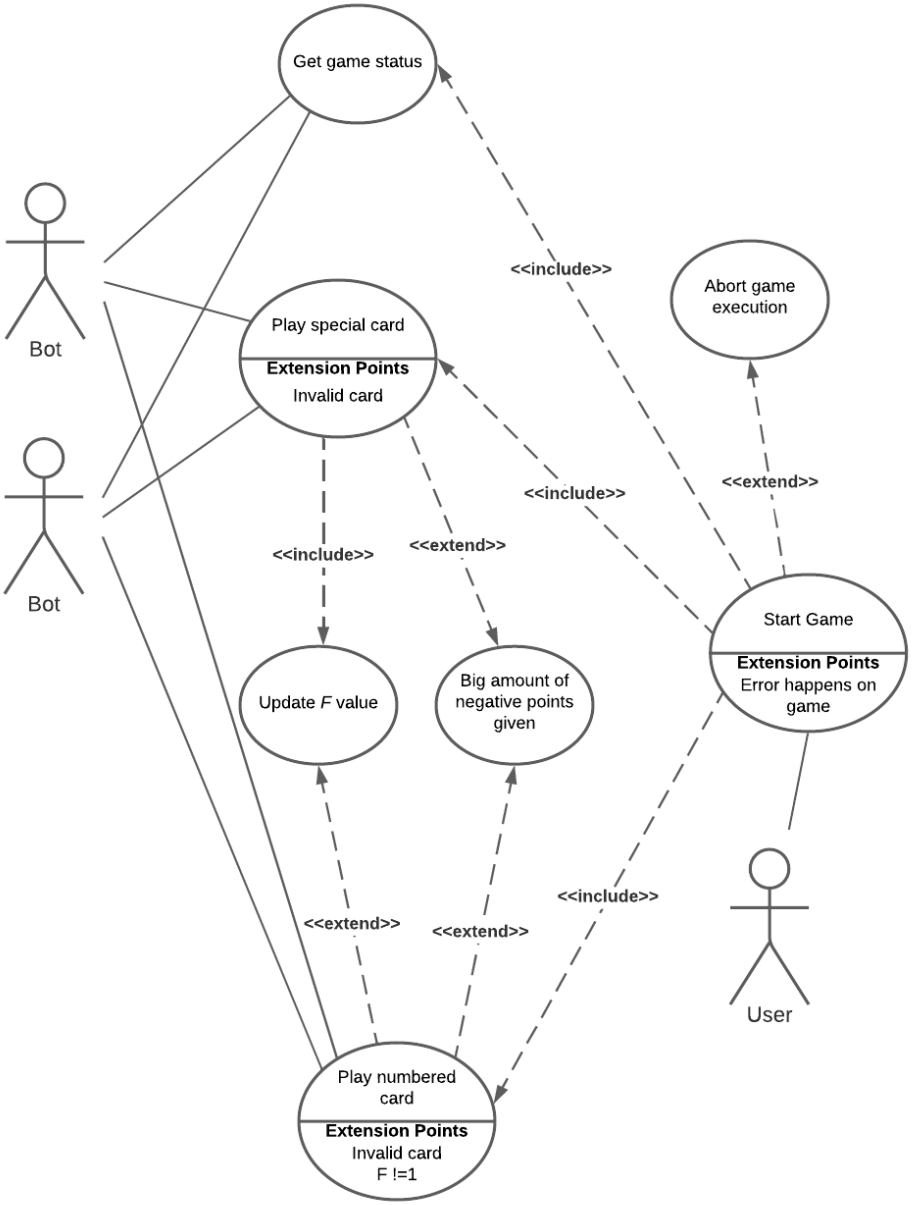


Figure 3.1: UML use case diagram for ASMACAG.

3.3.3 Software architecture

The code is architected to be easily understood and modified, with highly decoupled classes. Firstly, there are some classes that describe the conceptual elements within the game. They are defined as follows.

- ***Card:*** The *Card* class conceptualises the cards used in ASMACAG by using an *Enum* with the different special cards called *CardType*. If it is a numbered card it also holds a numeric value.
- ***Action:*** The *Action* class conceptualises a card being played. If the *Card* played is an special card it only needs to hold that card, otherwise it holds two *Card* instances. One of them is the one from the hand of the player and the other one is the *Card* on the board.
- ***CardCollection:*** The *CardCollection* class keeps an ordered list of *Card* instances and provides several methods to easily handle them. Is is flexible enough so that there is no need for a different class describing a hand or deck.

Making use of the above described objects there are two different classes that can keep track of the current game status. The first one is *GameState*. This class is the one used by the *Game* which runs the actual matches. The other one is *Observation* which is used to provide the bots only with the information they should be allowed to consult, randomising the rest of values. This class also provides a set of methods that will make bot programming easier. Examples of this are a method to randomise its hidden parts again, getting another possible state of the game, or a method to get a random *Action* that is currently valid to play.

The rules for the game are kept entirely decoupled from it by the using the *ForwardModel* interface, which has the methods needed to decide how a *GameState* (or *Observation*, so that the bots can also make use of it) will advance forward with the game. Similarly there is also the *Player* interface which is used to define a bot. It also needs to implement a single method that given an *Observation* decides what *Action* to play. Note that although *ForwardModel* and *Player* are conceptually defined as interfaces they need to be implemented as abstract base classes in Python. These have all the functionality needed from interfaces in this context.

Then there is the *Game* class itself, which essentially just acts as the mediator between the rest of the classes to make a match come together. It keeps the current *GameState*, it requests what *Action* to play from a *Player* (enforcing compliance with the budget of time) and it requests the *ForwardModel* to update the *GameState* using the given *Action*. Also, parameters for the game such as the number of cards per player hand or the numbers included in the numbered cards are kept decoupled from both the *Game* and the *ForwardModel* in a class called *GameParameters*. An instance of these parameters is passed to the *Game* when starting the match so that they are taken into account.

Finally, there is another interface provided along with ASMACAG but separated from the framework itself. This interface is *Heuristic* and it defines a set of methods

useful to evaluate how good is for the current *Player* a given *GameState*. This aims to make it easy to implement bots decoupled from the methods that assess performance, by allowing them to just use the interface to request evaluations of the status of the game a possible turn would lead to.

3.3.4 Class diagram

The UML class diagram displayed in *Figure 3.2* describes the structure of the framework architecture. It also includes the test bots needed for the experiments, which are further described later in this document. The architecture has been designed with a focus on simplicity of use and extensibility, as explained above.

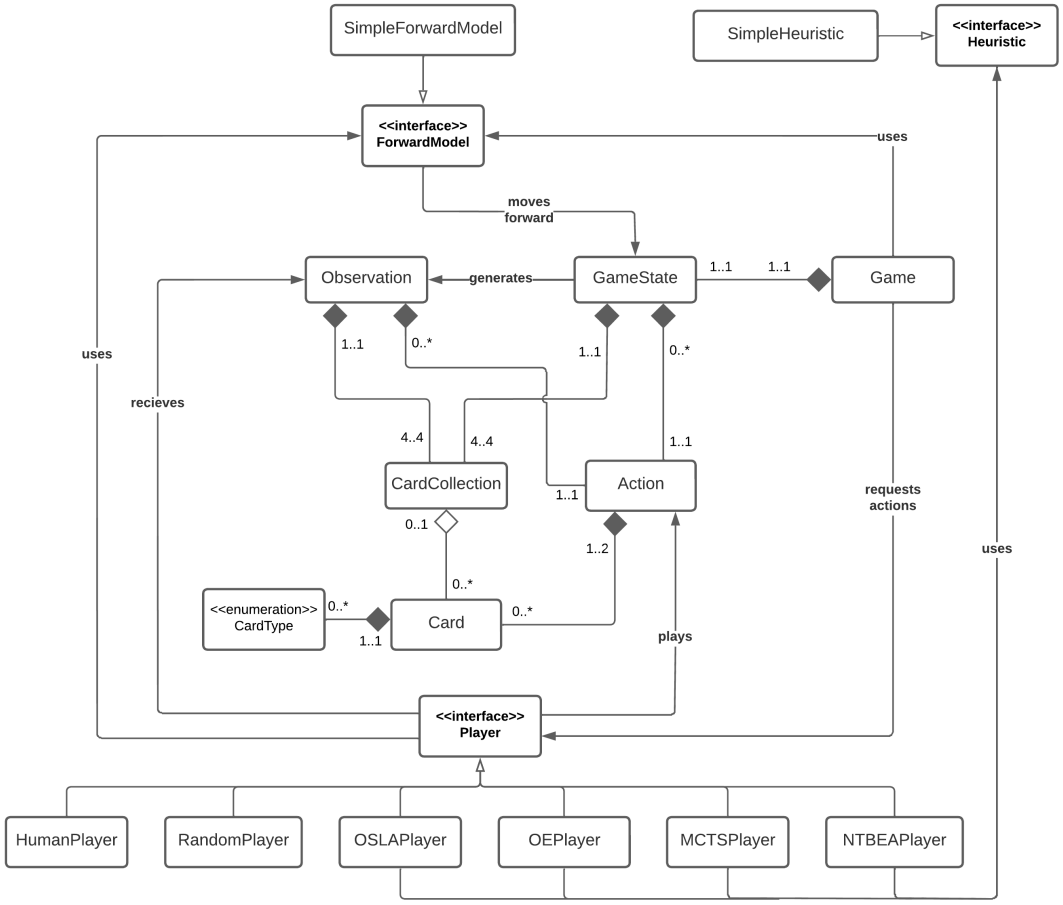


Figure 3.2: UML class diagram for ASMACAG.

3.4 System Architecture and Code Structure

ASMACAG, being a framework to run tests, is not tied to a specific system. The default values are adjusted to perform well with the computer described in *Section 2.2*, but these can be easily changed given the modular nature of ASMACAG.

Regarding code structure there are several best practices that ASMACAG takes heavily into account. The aim of these is to have a code that is easily readable, understandable and that allows the user of ASMACAG to quickly grasp anything they could need to. These practices are described bellow, together with the specific reasoning behind following them.

- **SOLID principles:** The SOLID programming principles are the Single responsibility Principle, the Open closed Principle, the Liskov Substitution Principle, the Interface Segregation Principle and the Dependency Inversion Principle. Following them leads to a code that is as decoupled as possible, as well as easily expandable. These principles are extremely well aligned with the design philosophy of ASMACAG as a programming tool and therefore are the foundations on which the codebase is laid out. They have been taken into account both when coding and when designing the software architecture previously described in *Subsection 3.3.3*.

- **PEP8:** The Python Enhancement Proposal 8 is a document that gives coding conventions for the Python code comprising the standard library in the main Python distribution [15]. It is community maintained and the most widely accepted guide on best practices on how Python code should be standardised so that it is readable and functional. In this regard, given the objective of ASMACAG of being easily understandable and simple for any programmer, adhering to it seems like a necessity.

- **Type hinting:** As Python is a dynamically typed language, understanding what kind of values are required for each method when modifying ASMACAG or programming a bot could be quite bothersome. For this reason, as well as for improving the documentation as explained below, the code for both ASMACAG and any bots implemented within the project uses type hinting for both the parameters and the return values of every single method. This way the user, specially when using a powerful IDE, can get coding faster and more easily.

- **Documentation:** When developing a package that should be easily used, having an extensive, but easily accessible and understandable, documentation is really important. For this reason every sub package, class and method is documented in ASMACAG. This is done with the use of docstrings. These docstrings provide all the needed information but do not fulfill the requirement of being accessible and usable. For this reason a proper documentation is generated from them following the process that will be described in *Subsection 4.1.4*.

3.5 UI and UX Design

There are two separate aspects to look into when talking about the user experience and the user interface of this project. The first one is how a programmer that uses ASMACAG as a tool interacts with the framework and the second one is the Human Player bot, which is a bot implementation that aims to allow a user to play ASMACAG, either against other user or against any bot. Therefore, this bot must take into consideration user experience and user interface aspects.

3.5.1 Use of the ASMACAG framework

ASMACAG, being a code framework doesn't have a proper interface itself, because it is ultimately just a code package. Even with this into consideration, there are some actions that have been taken with the intention of making the user experience of any developer that uses ASMACAG as smooth as possible. Most of them are as described in *Section 3.4*, such as providing a detailed documentation for everything in the framework or extensively using type hinting. In this regard a live web page version of the documentation is available online, as well as PDF version for offline use (see *Appendix B*).

Other measure taken for a better user experience with the framework is the generation of output files for the games to allow for easy storage of results. In this files there is also one of the most important and quality of life improving features for programmers using non deterministic algorithms on their bots. The file includes a seed that can be input to exactly reproduce that game, including bot decisions, allowing for easy step by step debugging of any previous game where anything might have gone wrong.

3.5.2 Use of Human Player

The Human Player bot is an implementation of the Player interface that allows the game to be played by a human. The user here acts as the player of the game and this allows the possibility of getting a deeper understanding of the way games take place and the actions that are undertaken by the bots. Given this approach, there are several considerations that have to be taken for the design of both the user experience and the user interface. It need to be a lightweight system, since its aim is to provide the relevant information as easily as possible, in an understandable and user-friendly way. It also need to accommodate the flexibility that ASMACAG allows to a certain extent, so that it doesn't limit the capabilities of the framework, and most importantly it has to accurately display the actions taken, so that they can be analysed in detail.

A graphical user interface is a possible approach to this. If used, all information can be displayed in a 2D graphical environment with simple graphics. It would display the cards on the table as well as the cards on each player hand. The players' name and score could be displayed close to its hand and highlighted during its turn. When a card is played it can be highlighted and complemented with some basic movement to show how it has been used. The graphical user interface could also show changes in each bot's score and display the value of F. In *Figure 3.3* a basic mockup of how the GUI could

be laid out is shown with the location of the cards, names, scores and the F value. Note how the current player is highlighted using red, to show whose turn it is.

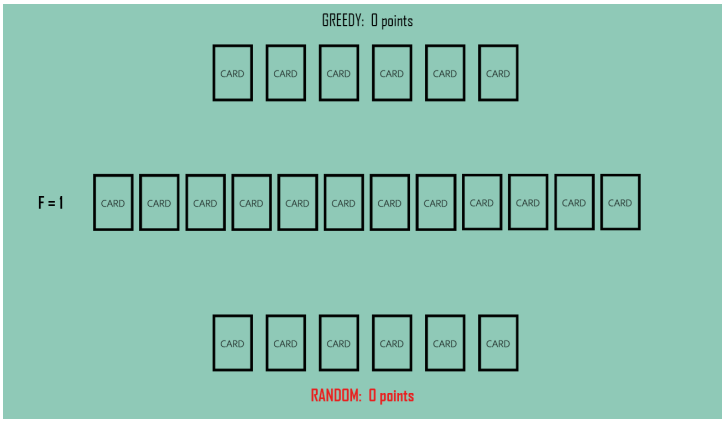


Figure 3.3: Proposed GUI design for ASMACAG.

Another way to accomplish the delivery of the information to the user, as well as allowing user input into the game could be trough a command line based interface. This can allow to show a more detailed information with all the needed data from the game. It also allows for a far greater flexibility in game parameters and game rules without major changes, while the graphical interface could need a big redesign for each game variation needed or even need to be modified after certain parameter changes. The proposed command line based user interface displays a list of possible actions that the user can execute, given the current observation, and lets the user input the index of the action they want to choose, a example of this is shown in *Figure 3.5*. This can be combined with executing the game with a *verbose* parameter, which can display the current game state per each turn as seen on *Figure 3.4*.

```
---------------------------------------
Player 1 [OSLAPlayer] turn
---------------------------------------
TURN: 1
BOARD:  [CardType.NUMBER 3] [CardType.NUMBER 6] [CardType.NUMBER 2]
        [CardType.NUMBER 5] [CardType.NUMBER 5] [CardType.NUMBER 3]
        [CardType.NUMBER 5] [CardType.NUMBER 6] [CardType.NUMBER 2]
HAND P1:    [CardType.NUMBER 3] [CardType.NUMBER 4] [CardType.MULT2]
            [CardType.NUMBER 5] [CardType.DIV2] [CardType.MULT2]
SCORE P1: 1.0
HAND P2:    [CardType.NUMBER 2] [CardType.NUMBER 3] [CardType.NUMBER 2]
            [CardType.NUMBER 1] [CardType.DIV2] [CardType.DIV2]
SCORE P2: 12
FACTOR: 1
ACTION POINTS LEFT: 3
```

Figure 3.4: Game state display for ASMACAG when using the *verbose* parameter.

With this analysis of the possible ways human interaction could be implemented into a bot, the decision to go forward with the command line version has been taken, which

```
Actions that can be played:
0->[CardType.MULT2] on [nothing]
1->[CardType.NUMBER 1] on [CardType.NUMBER 4]
2->[CardType.NUMBER 1] on [CardType.NUMBER 5]
3->[CardType.NUMBER 1] on [CardType.NUMBER 6]
4->[CardType.NUMBER 1] on [CardType.NUMBER 2]
5->[CardType.NUMBER 2] on [CardType.NUMBER 4]
6->[CardType.NUMBER 2] on [CardType.NUMBER 5]
7->[CardType.NUMBER 2] on [CardType.NUMBER 6]
8->[CardType.NUMBER 2] on [CardType.NUMBER 2]
9->[CardType.NUMBER 3] on [CardType.NUMBER 4]
10->[CardType.NUMBER 3] on [CardType.NUMBER 5]
11->[CardType.NUMBER 3] on [CardType.NUMBER 6]
12->[CardType.NUMBER 3] on [CardType.NUMBER 2]
13->[CardType.NUMBER 6] on [CardType.NUMBER 4]
14->[CardType.NUMBER 6] on [CardType.NUMBER 5]
15->[CardType.NUMBER 6] on [CardType.NUMBER 6]
16->[CardType.NUMBER 6] on [CardType.NUMBER 2]
Select the action index: 3
Player 1 selects [CardType.NUMBER 1] on [CardType.NUMBER 6].
Score: [11] - [-5]
```

Figure 3.5: Proposed command based UI design for ASMACAG.

yields an overall better user experience for the specific tasks it has to accomplish within this project. Even though the GUI might be a better user interface in some aspects, adopting it would sacrifice some of the core aspects of this project's philosophy, which places the user experience above user interface design when they present a conflict. The command line interface allows for far greater flexibility, which in this context is extremely valuable.

It is also acknowledged that the GUI could be included additionally to the command line interface. This would not have any additional drawbacks for the framework, because it could be provided as an additional package. The problem with this addition is that it would impact the time needed for the development of this step of the project significantly, probably doubling it, or even more. Given this impact, the graphical approach to the interface is deemed not suitable for the context of the project. Sticking to the time limitations established in the plan is important so that most of the time can be devoted to bot development and experiments, which are the core components of the project.

3.6 Algorithms implemented

This section describes the different algorithms to be implemented as bots and tested using ASMACAG during the project. It also goes through the reasoning behind their selection as part of the experiments that will follow their implementation.

3.6.1 Random

This algorithm will just choose randomly one of the available valid actions when required to decide, without looking into the next action to play at all. It will be implemented into the Random Player bot. It is included to be used as a minimal baseline, to check that each of the other algorithms are able to beat it in an statistically relevant and consistent way.

3.6.2 Greedy One-Step Lookahead

This algorithm will just perform a greedy search on the available valid actions at one moment of the turn, playing the one that returns the best value. Once again, it will not go through the next actions that could be played during that turn. This algorithm will be used by the OSLA Player bot. It is useful as a baseline heuristic algorithm, testing the capability of each of the other algorithms to win against it, when applying the same heuristic.

3.6.3 Monte Carlo Tree Search Algorithm

This algorithm is probably the most researched of the ones tested and it has been selected for its relevance in the AI field during the last decades. First coined in 2006 [4], it has been proposed almost since its conception as a useful algorithm for game AI [2]. It iteratively builds a tree that estimates the long term value of each action until the time budget is hit. Then it returns the estimated best performing action at that point. The nodes represent the state space of the game, while the edges represent the actions. Per iteration it applies the following 4 steps [1]:

1. **Selection:** From the root node, a child selection policy is applied to descend recursively through the tree to find the most urgent expandable node, given that a node is expandable if it represents a non-terminal state and has unvisited children. This policy is usually the calculation of a UCB value, which is what will be used to implement this bot.

2. **Expansion:** Child nodes are added, expanding the tree. In ASMACAG this implies getting the possible actions from that node and creating a new child node per action.

3. **Simulation (or rollout):** A simulation is run from the new nodes according to the default policy to produce an outcome. In ASMACAG the outcome would be the turn score and the simulation would be to select random actions from that point in the turn until the turn ends.

4. **Backpropagation:** The outcome is propagated through the transversed nodes, updating the nodes' statistics.

A basic summary of how this iterative process takes place can be seen on *Figure 3.6*. It represents one iteration of the process that would continue until the budget limit is hit (is the case that concerns us, when the time reaches the limit set for the turn calculation in the experiment). This algorithm will be implemented into the MCTS Player bot.

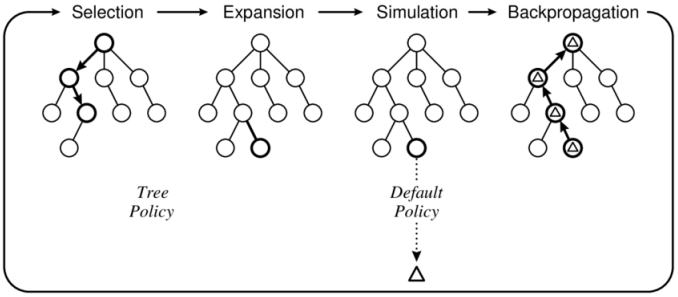


Figure 3.6: One iteration of the general MCTS approach. From [1].

3.6.4 Online Evolution Algorithm

This algorithm has been selected for it's relevance for the problem of multi-action adversarial games. First proposed in 2016 [6], it aims to solve this specific problem. The Online Evolutionary Planning algorithm is based on the concept of applying genetic algorithms by evolving actions during a game (online) instead of using evolution to train an agent that will play the game later (offline). This concept was first introduced, though applied to a different type of games, by the Rolling Horizon Evolution algorithm proposed at [12].

The Online Evolution algorithm tries to apply the aforementioned idea by using an evolutionary algorithm on a set of actions composing a turn. As it is stated by Niels Justesen in [6], "it can be seen as a single iteration of Rolling Horizon Evolution with a very short horizon (one turn)". The algorithm tries to find the best combination of actions for a turn by evolving an ordered set of actions, going through the following steps in a loop.

1. **Selection:** Evaluates each of the possible turns in the current population and kills the worst ones.

2. **Crossover:** Generates new possible turns to replace the killed ones by combining pairs of the ones that are still alive.

3. **Mutation:** In this step there is a certain probability that an action within some of the possible turns that were generated during the crossover process is replaced with a random one.

This algorithm focuses on the idea of solving the problem that is the most specific to multi-action games, which is finding the right combination of actions to compose a turn,

and therefore should be quite efficient when trying to win at ASMACAG. It will be implemented into the OE Player bot.

3.6.5 N-Tuple Bandit Evolutionary Algorithm

As a novel approach to solving multi-action adversarial games, the N-Tuple Bandit Evolutionary Algorithm is also proposed. This algorithm was first designed for its use in offline optimization of an heuristic function [8] and of the hyperparameters of an agent for solving a set of games [10]. The idea suggested is attempting to apply it to evolving in-game actions during the turn calculation, using a similar approach to the one Online Evolution uses to adapt traditional evolutionary algorithms.

The N-Tuple system in the algorithm directly models the statistics, while approximating the fitness and number of evaluations of each modelled combination of parameters [9]. This way, the algorithm places the focus on the combinations of the set of parameters being optimised. Applying it to multi-action decision making, the aim is to further focus on finding the right combinations of actions. Since this is the main problem of multi-action games, as discussed before, this approach could give interesting results for this specific kind of game.

The advantage it can have over other approaches, and specifically against other evolutionary algorithms, is that the N-Tuple based model in the algorithm is faster to access than the game itself. The algorithm tests several turns against that model, which approximates the reward based on the data received up to that point, per each time it tests a turn with the actual game. This allows the N-Tuple Bandit Evolutionary Algorithm to test a lot more possible turns in the same amount of time. This algorithm will be used by the bot named NTBEA Player.

3.7 Experiment Design

The experiments using the developed bots are an essential part of this work. They aim to enable justified conclusions to be taken from them and provide useful results and insights about each algorithms' behaviour. They should also prove ASMACAG's usefulness as a bot testing framework. The experimental methodology for the experiments performed at the end of this project is as follows.

The main experiment that articulates this project is performed by using 5 different bots to play ASMACAG. The five bots used are the ones presented above in *Section 3.6*. All of the bots will play 1000 games against each other. Each bot will be the first player for half of the games played against any other bot and the second player for the other half of them. This will allow to reduce the bias introduced by the fact that the game is easier to win as the first player. All bots will have a budget to make decisions which will be defined as time limit per turn, these time will be of 1 second.

Regarding the evaluation of the game states where an heuristic is needed all bots will make use of the same one, it being the score difference between the players. Finally, to evaluate the performance of each bot the number of wins will be used. The performance

results will need to be assessed on their statistical relevance to draw conclusions on how well each bot works.

Before the main experiment, there will be another experiment for adjusting the parameters of each bot that needs it. These bots are MCTS Player, OE Player and NTBEA Player. For each one of them a set of possible values per parameter will be chosen. Then 1000 games will be played against OSLA Player per each possible value for the parameter, keeping the rest of the parameters constant. This will allow to decide on the best parameter set for each bot, which will be then used on the main experiment explained above.

Chapter 4

Work Development and Results

Contents

This chapter of the report describes the process of developing the project and the results obtained.

4.1 Work Development

This section goes through each of the core aspects of the development process, as well as the way the experiments have been performed. It will follow a chronological order, going through each of the steps that include relevant information for the report.

4.1.1 Development Environment

The first step of the development process is to set up a proper development environment that allows for a quick and easy programming pipeline. There are several tools that were set up for the development of ASMACAG. The first thing to be set up was a local *Git* repository, with a remote hosted on *Github*. For managing this repository the tool *Fork* was set up. Its use has allowed to streamline the process of managing the repository, by providing features such as easy control of the different branches and stashes or simple ways to rebase comments interactively.

Next, the IDE was set up. The tool chosen was JetBrain's *Pycharm*, due to its ample feature set. Tools such as its powerful debugger proved to be very useful further down the development process. Another feature provided by this editor and extensively used

during the coding of ASMACAG is the possibility to get warnings for any breach of the PEP8 guidelines. This way, a readable and guideline-compliant code can be written without systematically having to refer to the corresponding Python Enhancement Proposal every single time. Finally, *Pycharm* also facilitates type hinting, which has been deemed pretty important for the development of this framework as explained in *Section 3.4*.

Another tool that was set up, in this case as a *Pycharm* plugin, is *Github Copilot*. This is a code auto completion tool that works using machine learning. Specifically, it is based on *Codex*, a GPT language model fine-tuned on publicly available code from *GitHub* [3]. This tool suggests code completions based on the rest of your code. It is specially useful to streamline repetitive tasks or to help auto complete docstrings for better documentation. Using this features the programmer can work quicker and be more focused on the important structural aspects of the code, while being somewhat less error prone.

4.1.2 Programming ASMACAG

Programming the main ASMACAG framework was a task that mainly encompassed following with precision the architecture, best practises and processes laid out in *Chapter 3*. It also involved making good use of the tools available in the development environment that was set up, as it has been just described in *Subsection 4.1.1*.

Once a first version of the framework was finished and it passed some basic unit tests the development process already moved on into the bot development. But after developing the first three bots described in *Subsection 4.1.3*, those bots were used to perform in depth debugging on the framework. After this debugging process, that yielded fixes for any errors produced during the development process, the project was ready to go on and continue again with the bot programming, so that the experiments could soon be run.

4.1.3 Programming the Bots

The bots were programmed in order from simpler to most complex, as seen below. A summary of any relevant elements of each of the bot's development process is provided here. All of these bots have been programmed by overriding the abstract base class *Player*.

1. **Random Player:** For this bot the only thing needed was to return a random *Action* from the *Observation* received.

2. **OSLA Player:** This bot just requests all the actions from the *Observation* and compares the effect of applying each of the using an *Heuristic*. It will not go through the turn, as it would run out of time, but just return the action that gives a bigger immediate reward.

3. **Human Player:** For this bot a command line interface as described on *Subsection 3.5.2* was developed. There were no major difficulties other than polishing the print statements so that they are easily readable, to make interaction simpler.

4. **MCTS Player:** The development of the MCTS-based bot started by creating the *MCTSNode* class. This class represents a node in the MCTS tree and contains the methods needed to manage its children. It also contains methods to perform each of the operations used for the algorithm, these are expansion, simulation and backpropagation. Then, the actual *MCTSPlayer* class just needs to hold the root node and follow the algorithm, going through the tree and performing the aforementioned operations accordingly.

5. **OE Player:** For this bot, firstly, a *TurnGenome* class was developed. This class holds a turn and act as a genome for the genetic algorithm. I holds methods for performing the actions such as crossovers or mutations. One problem that needed to be fixed during development was the fact that a turn can be no longer valid when performing any of this actions. For this reasons the methods need to use the forward model to actually test for validity on the new turn and replace any invalid actions. The algorithm itself was implemented in the *OEPlayer* class, making use of the *TurnGenome* class.

6. **NTBEA Player:** The development of this bot started with the development of the *Bandit1D* and *Bandit2D* classes. These were based on the ones available in the NTBEA implementation at [11]. This classes are used to compose the statistical model that the algorithm uses. They hold methods for adding new stats or requesting them. Then the *FitnessEvaluator* was programmed. This class is used to calculate the fitness of a turn decided by NTBEA. It needs to translate between an Action list and the integer list the *Bandit1D* and *Bandit2D* use, and the other way round. The N-Tuple Bandit Evolutionary Algorithm itself was implemented in the *NTBEAPlayer* class, making use of the aforemetioned classes and some helper methods for better organisation and readability.

Also note that all the bots with algorithms that are performed until the time runs out are implemented so that they calculate the whole turn using all of the budget when they are requested the first action. Then they just keep the next actions in a list and pop them when requested.

4.1.4 Creating the Documentation

During the whole development process the documentation started to be shaped, by adding docstrings and type hints to all methods, classes and package files. Then a revision of the whole code base was needed to fine tune it.

The next step was to generate the actual documentation files so that it can be navigated through easily and it provides value to the end user. For this the package and tool *pdoc3* was used. Since it takes the documentation from the previously written type

hints and docstrings not much extra work should be needed, though there were a couple problems to account for.

Firstly, there was the need to use forward typing in some cases, to prevent *pdoc3* from not identifying the class in a type hint when it had not parsed it yet. This is a feature from python that allows to reference a not yet defined name in type hints by using an string literal, as defined on [14].

The other thing to account for was the fact that *pdoc3* requires references to other parts of the documentation to be between backticks and fully qualified from the root of the project. This means that to reference a class, such as *Action*, from any part of the documentation you would need to write 'ASMACAG.Game.Action.Action' where *ASMACAG* is the root package, *Game* is a sub package and *Action* first refers to the file name and then to the class name. This required some rewriting of parts of the documentation.

Finally, the documentation could be generated both in a PDF version and an HTML version. The PDF version is provided in *Appendix B*. The HTML version was uploaded to the code repository in *AppendixA* to separate branch. Then that branch was set as the Github Pages source for the repository. This way the whole documentation is easily accessible and navigable as a live web page at `https://dvs99.github.io/ASMACAG/`.

4.1.5 Runnning Experiments

For running the experiments, the file *play_n_games.py* was created. It contains a main program that plays a set number of games between each pair of players given, in this case 1000 games. Then, the variables to adjust in each algorithm had to be identified and adjusted by testing their performance against the One Step Lookahead Algorithm. The whole set of experiments described bellow, including the final one, are available on the file *reproduce_experiments.py* so that they can be easily analysed and reproduced by anyone.

For the Monte Carlo Tree Search Algorithm the value of C in the UCB function had to be fine tuned. This is needed because the most common value of $\sqrt{2}$, which ensures asymptotic optimallity, only does so when rewards are in the [0,1] range [7]. The set of possible values to test was decided to be $C = \{0.5, 1.414, 3, 8, 14, 20, 30, 45\}$.

For the Online Evolution Algorithm three values needed to be adjusted. The first is the size of the population, this is the amount of *TurnGenome* per generation of the algorithm. For testing this parameter the other two values where kept constant at *survival rate = 0.15* and *mutation rate = 0.35* while it was tested for the set of values *population size = {25, 75, 125, 175}*. Then the mutation rate was adjusted. This parameter controls the probability of mutating each *Action* of a *TurnGenome* after generating it from its parents crossover process. For this, the other values were kept constant at *population size = 75* and *survival rate = 0.35* while the bot was tested for the set of values *mutation rate = {0.05, 0.15, 0.25, 0.35}*. Finally, the survival rate had to be adjusted. This is the percentage of *TurnGenome* that will survive after a generation and will be used to generate the new ones. During the process, the other

values were kept constant at *population size = 75* and *mutation rate = 0.15* while the test was performed for the set of values *survival rate = {0.05, 0.15, 0.35, 0.55, 0.75}*.

The next bot to be adjusted was the one using the N-Tuple Bandit Evolutionary Algorithm. Firstly, it was decided that the value for C in the UCB function was going to be taken from the adjustment of MCTS Player. That value would only need further adjustments if the scores returned by the game were in a different value range, which is not the case. Then, the amount of neighbours per iteration needed to be adjusted. This value controls the amount of neighbours of the current turn that are compared against the NTBEA model to then choose the best one as the new current. In this context a neighbour is a turn with one *Action* different from the current one. Note that other actions may be different too, depending on the next parameter. For the experiment the other values were kept constant at *mutation rate = 0.3* and *initialization amount = 500*. The set of values tested was *neighbour amount = {5, 10, 20, 50, 120}*. Next, the mutation rate was adjusted, being the probability of any extra *Action* changing when generating the current turn's neighbours. During the process, the rest of the parameters stayed constant at *neighbour amount = 20* and *initialization amount = 500*. The bot was tested for the set of values *mutation rate = {0.05, 0.15, 0.3, 0.55, 0.8}*. Lastly, the adjustment of the initialization amount was made. This parameter sets how many turns will be initially evaluated using the *ForwardModel*, to then select the best of them as the current initial turn for NTBEA. For testing this parameter the other ones where kept constant at *neighbour amount = 20* and *mutation rate = 0.3*, while the bot was tested for the set of values *initialization amount = {50, 100, 500, 1000, 3000}*.

When all the parameters had been appropriately adjusted to their optimal values from the given sets, the final experiment was run. This involved testing each of the bots, with the decided parameters, against all other bots.

4.2 Results

This section discusses the results of the project and whether they are in line with the objectives laid out at the start of its development. To do this it goes through the main aspects of the project.

4.2.1 ASMACAG

The first of the outcomes from this project's development is the ASMACAG framework. It has ended up accomplishing the objectives it was aiming to. It is a well documented framework, that keeps thing simple. But, as it can be seen from the experimental results presented later in *Subsection 4.2.2*, it can reliably produce significant results about the performances of the different bots.

The flexibility and modularity objectives have also been accomplished, as it is evidenced by the easiness that the tool provides when anything needs to be modified. This allows ASMACAG to be a really useful framework for both research and education, specially when combined with the good documentation and simplicity mentioned above.

4.2.2 Experimental results

The first experiment to be performed was the value adjusting for MCTS Player. The results are displayed in *Table 4.1* and *Figure 4.1*. Please, note that the axes in the chart are truncated to better display the changes between different values of C. As it can be seen in the data, the performance of MCTS Player when playing against OSLA Player peaks in a value of $C = 8$, which was used for the final experiment.

Table 4.1
Results table from the experiment for adjusting MCTS Player's parameters

C value	*MCTS wins*	*OSLA wins*	*Ties*
0.5	502	485	13
0.141	509	473	18
3	523	458	19
8	545	436	19
14	520	470	10
20	507	478	15
30	477	507	16
45	458	530	12

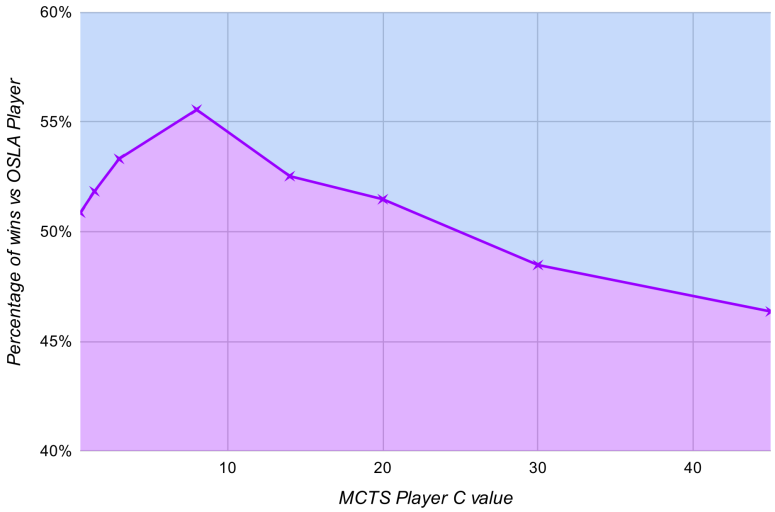


Figure 4.1: Results chart from the experiment for adjusting MCTS Player's parameters.

The next experiment to be run was aimed to adjusting the parameter values for OE Player. The results are displayed in *Table 4.2*. To show the results clearly, charts separated per the parameter being adjusted are provided in *Figure 4.2*, *Figure 4.3* and *Figure 4.4*. Please, note that the axes in the charts are truncated to better display the changes between different values of each parameter, and the axis values are different

from the previous experiments. Also take into account that the first row of the table is used in all three charts, as the information it holds is relevant for all three parameters' adjustment. As it can be seen in the data, the performance of OE Player against OSLA Player peaks in the following values: *population size = 125*, *mutation rate = 0.15* and *survival rate = 0.15*. The combination of those parameters results in the bot used for the final experiment.

Table 4.2
Results table from the experiment for adjusting OE Player's parameters

Population size	*Mutation rate*	*Survival rate*	*OE wins*	*OSLA wins*	*Ties*
75	0.15	0.35	695	292	13
25	0.15	0.35	669	314	17
125	0.15	0.35	704	286	10
175	0.15	0.35	692	292	16
75	0.05	0.35	680	302	18
75	0.25	0.35	678	303	19
75	0.35	0.35	675	308	17
75	0.15	0.05	688	302	10
75	0.15	0.15	700	283	17
75	0.15	0.55	671	306	23
75	0.15	0.75	675	311	14

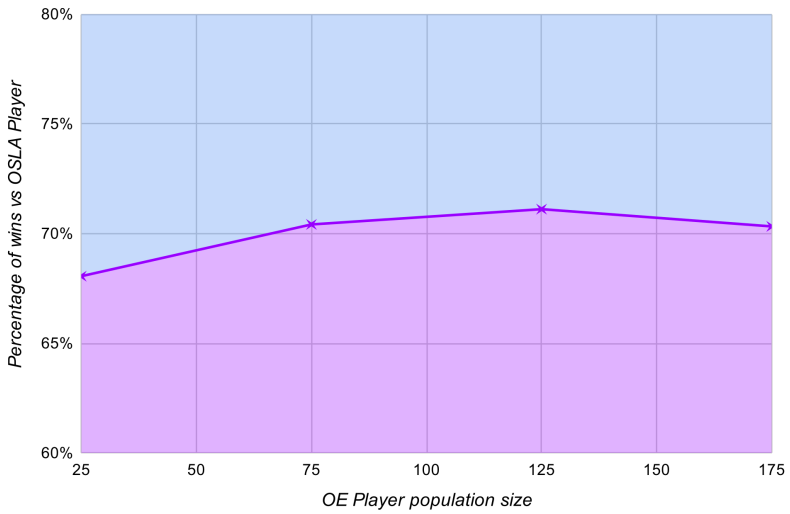


Figure 4.2: Results chart from the experiment for adjusting OE Player's population size.

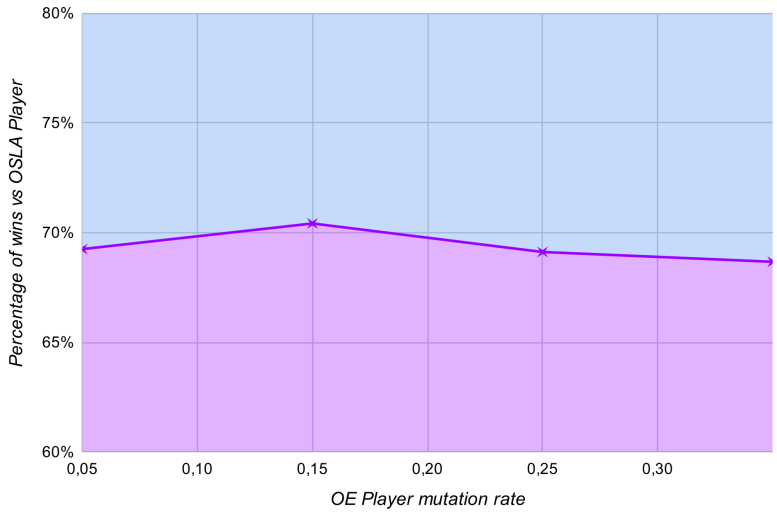


Figure 4.3: Results chart from the experiment for adjusting OE Player's mutation rate.

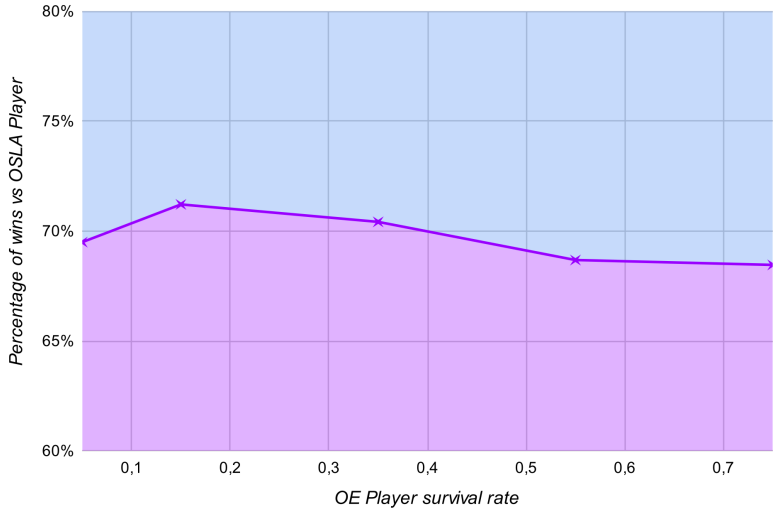


Figure 4.4: Results chart from the experiment for adjusting OE Player's survival rate.

The last of the parameter-adjusting experiments was aimed at deciding the values to be used for NTBEA Player. The results are displayed in *Table 4.3*. To show the results clearly, charts separated per parameter being adjusted are provided in *Figure 4.5*, *Figure 4.6* and *Figure 4.7*. Once again, note that the axes in the charts are truncated to better display the changes between different values of each parameter and take also into account that the first row of the table is used in all three charts, as the information it holds is relevant for all parameters' adjustment. The performance of NTBEA Player when playing against OSLA Player peaks in the following values: *neighbour amount* = *5*,

mutation rate = 0.55 and *initialization amount = 1000*. The neighbour amount shows a clear trend towards the selected value, or an slightly lower one, being optimal. The other two parameters seem to show a more unclear trend, but nevertheless the combination of those peak values is used for the NTBEA Player in the final experiment.

Table 4.3
Results table from the experiment for adjusting NTBEA Player's parameters

Neighbours	*Mutation rate*	*Initializations*	*NTBEA wins*	*OSLA wins*	*Ties*
20	0.3	500	721	268	11
5	0.3	500	733	250	17
10	0.3	500	733	254	13
50	0.3	500	668	309	23
120	0.3	500	569	410	21
20	0.05	500	730	248	22
20	0.15	500	731	257	12
20	0.55	500	772	216	12
20	0.8	500	727	254	19
20	0.3	50	736	245	19
20	0.3	100	733	249	18
20	0.3	1 000	744	244	12
20	0.3	3 000	720	266	14

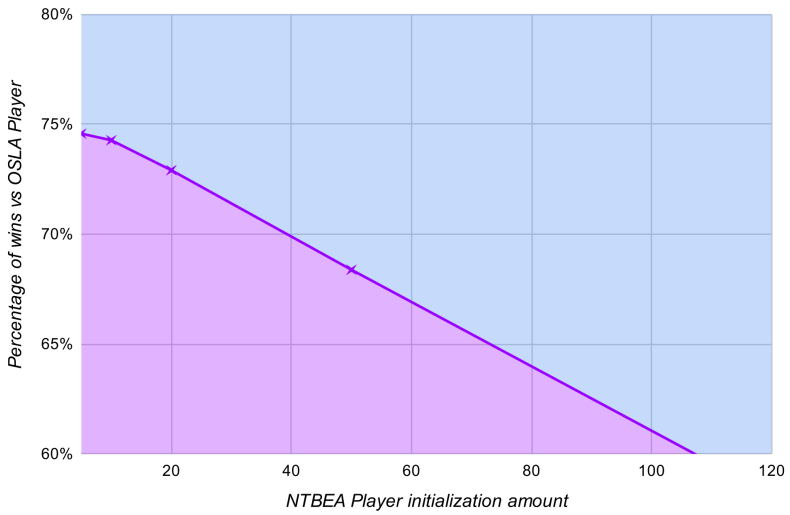


Figure 4.5: Results chart from the experiment for adjusting NTBEA Player's neighbour amount.

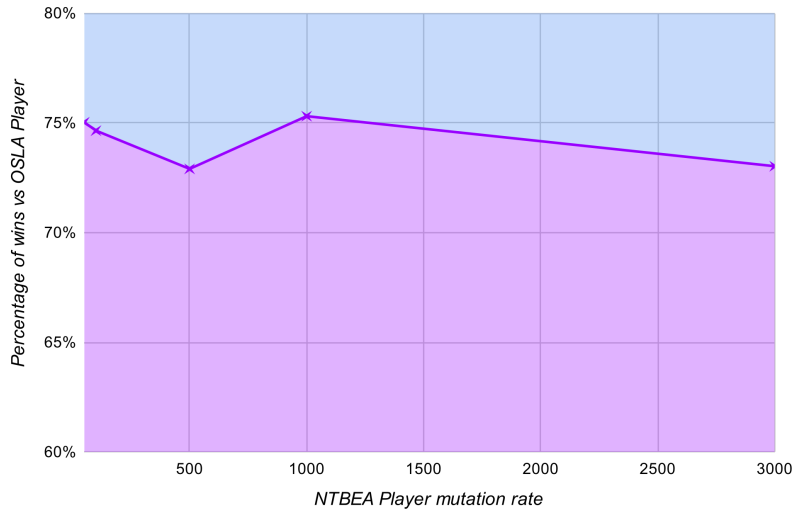


Figure 4.6: Results chart from the experiment for adjusting NTBEA Player's mutation rate.

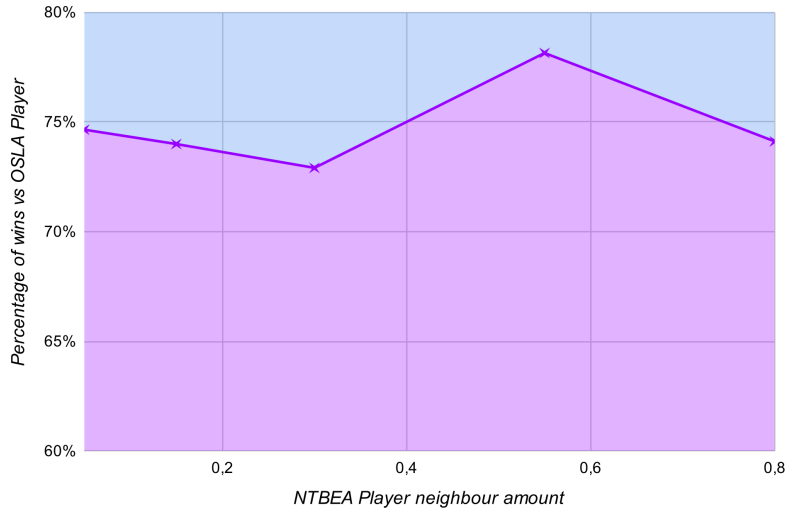


Figure 4.7: Results chart from the experiment for adjusting NTBEA Player's initializations.

After finishing the experiments for parameter adjusting, the final experiment could be performed. The parameters used were the best found for each of the bots. The results are shown using *Table 4.4* and *Figure 4.8*. Random Player loses almost every game, as it was expected. Given that its only use was to prove that the other algorithms were actually working, a version of the results disregarding the games against Random Player is provided in *Table 4.5* and *Figure 4.9*. There, the results are shown more clearly and without including ties, which also have a negligible effect in the overall results.

Table 4.4
Results table from final experiment

Player	*Wins*
Random	99
OSLA	1 896
MCTS	2 111
OE	2 726
NTBEA	3 035
Tie	133

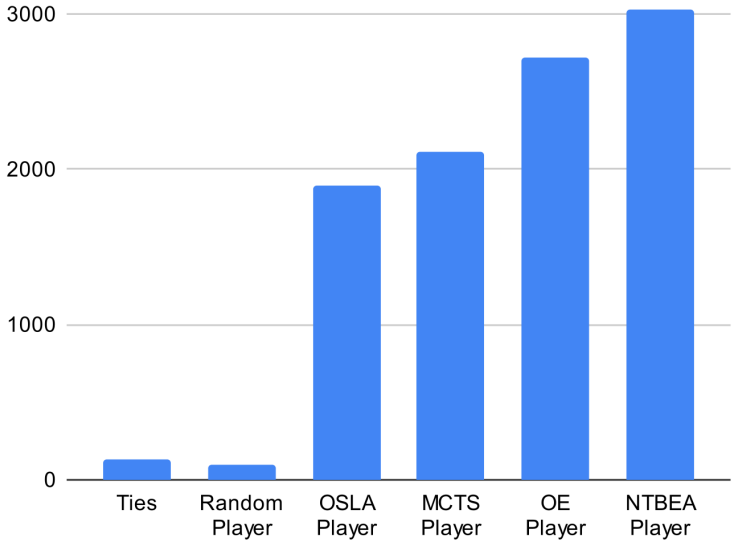


Figure 4.8: Results chart from final experiment.

Table 4.5
Results table from final experiment (except games with Random and ties)

Player	*Wins*
OSLA	940
MCTS	1 169
OE	1 738
NTBEA	2 047

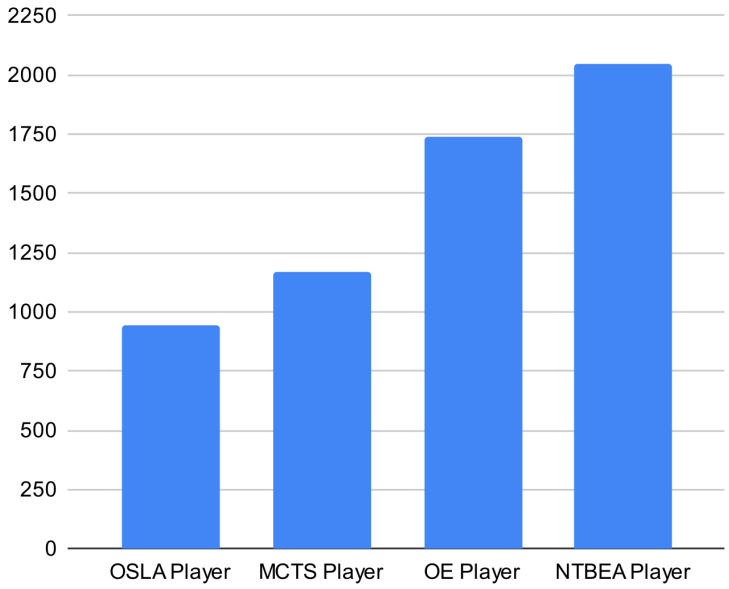


Figure 4.9: Results chart from final experiment (except games with Random and ties).

It can be seen that OSLA Player is the least performant bot, which is the expected result given that it doesn't make any attempt on finding a good combination of actions for the whole turn. Then MCTS Player follows relatively closely, showing that it can find combinations of actions but it is not as efficient as the evolutionary algorithms in doing so. Then both OE Player and NTBEA Player are quite ahead, because they use their evolutionary aspect to very efficiently find good combinations of *Actions*. Within them, NTBEA seems to have a not extremely big but clearly significant advantage. This proves that this novel approach to multi-action game solving, proposed on this project, can yield better results than the current state-of-the-art algorithms.

4.2.3 Project-wide results

This project has yielded some overall results, other than the ones already mentioned. One of them is this report describing the whole project vision and its development process.

Also, an academic communication that complements this report has been produced, in collaboration with Raúl Montoliu. It is titled *NTBOE: A new algorithm for efficiently playing multi-action adversarial games*. This communication has been sent to the *I Spanish Video Game Conference* organised by *SECiVi* and is currently under peer review process.

CHAPTER 5

Conclusions and Future Work

Contents

This is an opinion chapter aimed at giving closure to the project. It presents the personal conclusions from the student and it outlines what are the future prospects for ASMACAG and for the research process that the project has started.

5.1 Conclusions

The completion of this project has been a really successful endeavour. It has allowed me to personally research artificial intelligence applied to video games in great depth. It has also shown me that artificial intelligence has a really interesting future ahead when applied to video games.

The results that the project has yielded are quite interesting. They show the real possibilities of ASMACAG and, specially, the potential of applying a modified version of the N-Tuple Bandit Evolutionary Algorithm to efficiently playing multi-action adversarial games. This topic has been further researched to produce the paper mentioned in *Subsection 4.2.3* but it still has a lot of potential for in depth research, as explained in *Subsection 5.2*. I am very motivated to continue researching this and other similar topics.

Finally, I think that going though this work has allowed me to get experience on developing a software product in a quite professional manner. I have gone trough all the steps needed and carefully engineered ASMACAG and every bot to the best of my

ability, within the given time constraint. This has been a new experience and I think it has value both personally and professionally. On the other hand it has also allowed me to further experience research and academics. This has motivated me a lot to have professionally researching video game related topics and participating in academia in general as a personal goal.

5.2 Future work

Regarding ASMACAG as a framework and tool there is much more work that could be done, such as developing a graphical user interface as discussed in *Subsection 3.5.2*. But the work to do is not limited to this project's goals or to me as a researcher with specific interests. Its design has aimed to provide a tool that can be extended and used for educational or research purposes by anyone and that is the future envisioned for ASMACAG. The hope is that its open source code can be extended and used by anyone that needs it.

There is also the obvious possibility to develop more bots for further researching artificial intelligence applied to multi-action adversarial games. Likewise, developing new sets of rules for the experiments would also be really interesting. I will probably work on both of those things in the future.

Finally, the main branch of research and possible future work that is available after this project is further investigating the potential of applying to multi-action game solving the variation of the N-Tuple Bandit Evolutionary Algorithm used in NTBEA Player, either in the same way that has been done during this work or in a similar one. This is a broad topic that includes testing the consistency of the results with different parameters and rules, as well as implementing this same algorithm in several other multi-action games. If it is successfully executed this can be a really amazing path to explore and further research and it can lead to results that help move forward the field of artificial intelligence applied to multi-action adversarial games.

Bibliography

[1] Cameron Browne, Edward Powley, Daniel Whitehouse, Simon Lucas, Peter Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez Liebana, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4:1:1–43, 03 2012.

[2] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-carlo tree search: A new framework for game ai. In *Proceedings of the Fourth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, AIIDE'08, page 216–217, 2008.

[3] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, and Greg Brockman et al. Evaluating large language models trained on code. *ArXiv*, abs/2107.03374, 2021.

[4] Rémi Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In Paolo Ciancarini and H. Jaap van den Herik, editors, *5th International Conference on Computer and Games*, Turin, Italy, May 2006.

[5] Indeed. Junior programmer salaries. `https://es.indeed.com/career/programador-junior/salaries`. Accessed: 2022-06-10.

[6] Niels Justesen, Tobias Mahlmann, and Julian Togelius. Online evolution for multi-action adversarial games. In *EvoApplications*, 2016.

[7] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *ECML*, volume 4212 of *Lecture Notes in Computer Science*, pages 282–293. Springer, 2006.

[8] Simon Lucas, Jialin Liu, Ivan Bravi, Raluca Gaina, John Woodward, Vanessa Volz, and Diego Perez Liebana. Efficient evolutionary methods for game agent optimisation: Model-based is best, 2019.

[9] Simon M. M. Lucas, Jialin Liu, and Diego Perez Liebana. The n-tuple bandit evolutionary algorithm for game agent optimisation. *2018 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–9, 2018.

[10] Raúl Montoliu, Raluca D. Gaina, Diego Pérez-Liebana, Daniel Delgado, and Simon Lucas. Efficient heuristic policy optimisation for a challenging strategic card game. In *Applications of Evolutionary Computation: 23rd European Conference, EvoApplications 2020, Held as Part of EvoStar 2020, Seville, Spain, April 15–17, 2020, Proceedings*, page 403–418, Berlin, Heidelberg, 2020. Springer-Verlag.

[11] Raúl Montoliu, Toni Pérez-Navarro, and Joaquín Torres-Sospedra. Source code of IPIN2022 paper: "Efficient tuning of knn hyperparameters for indoor positioning with N-TBEA". https://github.com/montoliu/MontoliuPerezTorresIPIN22, 2022.

[12] Diego Pérez, Spyridon Samothrakis, Simon Lucas, and Philipp Rohlfshagen. Rolling horizon evolution versus tree search for navigation in single-player real-time games. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, GECCO '13, page 351–358, New York, NY, USA, 2013. Association for Computing Machinery.

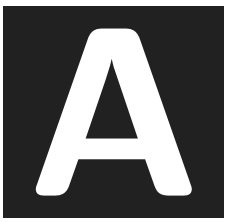
[13] Diego Villabrille Seca and Raúl Montoliu. ASMACAG: A new multi-action card game for facilitating the study of AI techniques. In *Proceedings of DiGRAES21 El Mapa y el Juego: los Game Studies en España*, 2021.

[14] Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa. Pep484. Type hints. Forward references. https://peps.python.org/pep-0484/#forward-references, 2001. Accessed: 2022-06-13.

[15] Guido van Rossum, Barry Warsaw, and Nick Coghlan. Pep8. Style guide for Python code. https://peps.python.org/pep-0008/, 2001. Accessed: 2022-06-12.

[16] Georgios N. Yannakakis and Julian Togelius. *Artificial Intelligence and Games*. Springer Publishing Company, 1st edition, 2018.

Appendix



Source code

The whole source code for the project is available on a *Git* repository hosted at `https://github.com/dvs99/ASMACAG`, including the experiments' results too. Since it is already available there with fully public access no code snippets are provided in this document. Please refer to the repository where you can check any part of the code as needed. Also, if you just want to navigate the structure or check an specific method you can do it using the documentation provided in *Appendix B*.

Appendix B

Documentation

The project documentation is available in two formats. The recommended way of accessing it is using the live HTML web page containing all the information. This page provides a well organised interface with links between any related parts for easy navigation. It can be visited at `https://dvs99.github.io/ASMACAG/index.html`. The other available format is a PDF document. It is not as feature-rich as the web version, but it still provides all the information in a format that is easy to store offline if needed. It can be downloaded at `https://raw.githubusercontent.com/dvs99/ASMACAG/master/docs/docs.pdf`.