



UNIVERSIDAD DE TALCA
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA EN DESARROLLO DE
VIDEOJUEGOS Y REALIDAD VIRTUAL

Design and Implementation of an Agent for the Planet Wars AI Challenge

ALVARO FRANCISCO CHOCOBAR BELMAR

Profesor Guía: JOSÉ DEL SOLAR ZAVALA

Profesor Co-guía: NICOLAS BARRIGA

Memoria para optar al título de
Ingeniero en Desarrollo de Videojuegos y Realidad Virtual

Talca – Chile
July 13, 2026



UNIVERSIDAD DE TALCA
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA EN DESARROLLO DE
VIDEOJUEGOS Y REALIDAD VIRTUAL

Design and Implementation of an Agent for the
Planet Wars AI Challenge

ALVARO FRANCISCO CHOCOBAR BELMAR

Profesor Guía: JOSÉ DEL SOLAR ZAVALA

Profesor Co-guía: NICOLAS BARRIGA

Profesor Informante: CRISTIAN VIDAL

Memoria para optar al título de
Ingeniero en Desarrollo de Videojuegos y Realidad Virtual

El presente documento fue calificado con nota: _____

Talca – Chile

July 13, 2026

Dedicado a mi familia, quien me apoyó y ayudó durante todo mi proceso universitario, así como a mis compañeros con quienes compartí tanto momentos difíciles como alegres durante estos 5 años.

ACKNOWLEDGMENTS

Agradecimientos a mis profesores, quienes me formaron durante mi vida universitaria, al profesor Nicolás Barriga, quien me orientó y enseñó sobre la importancia de la inteligencia artificial en los videojuegos, a mis compañeros y amigos con los que trabajé junto en las clases que tuvimos. Agradezco a mis padres por darme la oportunidad de estudiar lejos de mi hogar, a mi hermano menor, quien me acompañó y alentó durante este tiempo. Y me agradezco a mí por continuar y terminar esta etapa de mi vida.

TABLE OF CONTENTS

	Page
Dedicatory	i
Acknowledgments	ii
Table of Contents	iii
List of Figures	v
List of Tables	vi
Resumen	vii
Abstract	viii
1 Introduction	1
2 State of the Art	3
2.1 Planet Wars AI Challenge	4
2.2 Methods	5
2.2.1 Heuristic and Simulation Methods	5
2.2.2 Evolutionary and Optimization Methods	6
2.2.3 Reinforcement Learning Methods	7
2.2.4 Multi-agent Systems and Coordination	7
3 Methodology	8
3.1 Research Paradigm	9
3.2 Design Science Research Framework	9
3.3 Experimental Design	10
3.3.1 Environment Configuration	11
3.3.2 Baseline Agents	11
3.3.3 Metrics	12
3.3.4 Implementation Plan	12

4	Development of the ISILab	
	Agent	14
4.1	Architecture	15
4.1.1	Monte Carlo Tree Search	15
4.1.2	GameState Wrapper	17
4.2	Hyperparameter optimization	19
4.3	Versions of the ISILab agent	20
5	Evaluation and results	22
5.1	Experimental Setup	22
5.2	Results	23
5.3	Evaluation	27
6	Conclusion	29
	Bibliography	30
:	Appendix	35
7.1	IRace installation	35

LIST OF FIGURES

	Page
2.1 Screenshot of Planet Wars AI Challenge	5
3.1 Gantt Diagram 2025	13
3.2 Gantt Diagram 2026	13
4.1 Monte Carlo Tree Search.	16

LIST OF TABLES

	Page
5.1 Win rate of ISILab Agent versions against repository agents	24
5.2 Performance metrics of ISILab Agent versions	24
5.3 Win rate between ISILab Agent versions (100 games per matchup) . .	25
5.4 Summary of the best version of the agent from the competition . . .	26
5.5 Fisher’s Exact Test – Repository agents	27
5.6 Fisher’s Exact Test – Versions	27

RESUMEN

Los entornos de Real-time Strategy (RTS) son un gran espacio de investigación para agentes de inteligencia artificial (IA) gracias a la toma de decisiones en tiempo real y la variedad de acciones estratégicas que se pueden realizar. Existen una gran variedad de juegos RTS, cada uno de ellos con sus propias reglas y dificultades, otorgando un desafío para diseñar un agente competitivo. Para este proyecto se utiliza Planet Wars como espacio de prueba, donde nuestro agente debe administrar sus recursos y tomar decisiones tanto de defensa como de ataque en un período de tiempo limitado, además de enfrentarse a distintos escenarios cuyas reglas y parámetros van cambiando con el tiempo.

Este trabajo presenta el proceso de desarrollo de un agente competitivo para entornos RTS, evidenciando métodos eficientes para utilizar en futuras competencias tales como Planet Wars u otras investigaciones relacionadas con la toma de decisiones en tiempo real, pasando desde el estado del arte y explicando enfoques utilizados en juegos RTS como técnicas heurísticas, métodos de optimización, enfoques de aprendizaje por refuerzo y modelos de coordinación de múltiples agentes. El desarrollo de este proyecto se guía basándose en la metodología Design Science Research (DSR), la cual otorga un enfoque iterativo, llegando a crear 3 versiones del mismo agente, el cual utiliza el algoritmo Monte Carlo Tree Search (MCTS), mezclando 3 métodos, mostrando la evolución de su rendimiento en cada iteración.

Los resultados obtenidos dentro de la competencia son satisfactorios al ser nuestra primera participación en el Planet Wars AI Challenge, alcanzando la posición 17 dentro de los 30 agentes presentes, demostrando una buena estabilidad y desempeño competitivo frente a agentes evolutivos, avariciosos y superando con creces a todos los agentes random.

ABSTRACT

Real-time Strategy (RTS) environments are a great research space for artificial intelligence (AI) agents, thanks to their real-time decision-making and the variety of strategic actions available. There is a great variety of RTS games, each with its own rules and difficulties, providing a challenge for designing a competitive agent. For this project, Planet Wars serves as a testing space where our agent must manage its resources and make both defensive and offensive decisions within a limited time, while facing different scenarios whose rules and parameters change over time.

This work presents the development process of a competitive agent for RTS environments, evidencing efficient methods to use in future competitions such as Planet Wars or other research related to real-time decision-making, going from the state of the art and explaining approaches used in RTS games such as heuristic techniques, optimization methods, reinforcement learning approaches, and models of coordination of multiple agents. The development of this project is guided by the Design Science Research (DSR) methodology, which grants an iterative approach to create 3 versions of the same agent, which uses the Monte Carlo Tree Search (MCTS) algorithm, mixing 3 methods, showing the evolution of its performance in each iteration.

The results obtained in the competition are satisfactory, given our first participation in the Planet Wars AI Challenge. Ranking 17th among the 30 agents present, demonstrating good stability and competitive performance against evolutionary and greedy agents, and greatly surpassing all random agents.

1. Introduction

Real-time strategy (RTS) games present a major challenge for artificial intelligence (AI) research due to the combination of large action spaces, dynamic environments, and limited time for decisions [1]. Each scenario offers a wide range of possible actions, such as attacking or defending, influenced by changing rules and variables that shape strategies and adaptability. An intelligent agent, in this context, is a software entity capable of understanding the environment in which it is played and making strategic decisions to achieve specific results [2]. In RTS games; agents must balance exploration and exploitation to select actions that maximize long-term performance while adapting to constantly changing conditions [3].

These games offer complex scenarios that require real-time decision-making, demanding effective decision-making models, search strategies, adaptive heuristics, and learning algorithms. Platforms such as StarCraft [4] have served as important test environments for AI; however, they require significant computational resources and engineering efforts. For this reason, smaller environments, such as microRTS, have become popular alternatives [5]. These competitions, which do not require extensive computational resources, facilitate AI research by providing standardized, reproducible, and fast experimental environments.

The Planet Wars AI Challenge is a fast-paced strategic game competition on the Fast Planet Wars platform, where multiple AI agents compete against pre-designed opponents or those created by other participants. This competition serves as an ideal benchmark for AI research, providing an solid environment for planning, resource management, and tactical decision-making [6]. Thanks to its efficient implementation, the engine of Planet Wars runs at exceptional speed, allowing agents to test and execute a large number of strategies in a short amount of time.

In this project, we analyze these paradigms and evaluate the methods and algorithms used in designing intelligent agents for the Planet Wars AI Challenge and similar environments by reviewing heuristic methods, evolutionary algorithms, learning techniques, and multi-agent coordination approaches to identify the most effective strategies for addressing dynamic environments through adaptive and competitive behavior.

General Objective: Develop an intelligent agent capable of competing in the Planet Wars AI Challenge.

Specific Objectives

1. Review the state of the art in algorithms and AI techniques applied to RTS and microRTS environments, identifying their relevance for Planet Wars.
2. Design a modular agent architecture integrating heuristic, evolutionary methods, reinforcement learning, and multi-agent mechanisms.
3. Implement the agent within the Planet Wars environment using controlled experiments in the official competition repository.
4. Evaluate the agent's performance through quantitative metrics and compare it with baseline data and previous competitive agents.

To embark on this project, we will follow the research question: *What AI models and techniques used in RTS games can effectively improve an agent's performance in the Planet Wars AI Challenge?*

In the following pages, we analyze the state of the art applied to agents participating in previous Planet Wars AI Challenge competitions, along with research on methods for RTS games. This is followed by the methodology chapter, in which we define the approach we are taking to address this work and how the AI agent is evaluated to determine the best approach to develop for the competition. Afterward, we present the agent's development, explaining its functionality and the methods used in its creation. The next chapter covers the evaluation and results of the AI agent sent to the competition, analyzing its performance during the challenge and the repository evaluation. Finally, the last chapter is the conclusion of our work, in which we assess whether the agent's performance is robust.

2. State of the Art

Many agents and algorithms have been used over the years since this competition was established. This section covers the state of the art in the development of AI agents, investigating and analyzing methods previously used in the Planet Wars environment to identify the best possible options for addressing this competition.

Real-time strategy (RTS) games have been present in the world of video games since the 1990s, with Dune II (1992) ¹ being considered the defining game of the genre. Since then, this type of game has only evolved, delivering great titles such as Age of Empires ² and StarCraft ³. Over the years, this genre has become an ideal space for AI research [7] because it offers partially observable environments that provide ample scope for action and long-term strategic planning, thereby presenting challenges for the agent’s adaptability and resource management. These properties have become attractive for hybrid architectures and multi-agent methods [7] [8].

While RTS environments have proven ideal for AI research, large-scale games like StarCraft often incur high computational costs, which pose a barrier for researchers without access to high-performance resources. For example, the AlphaStar system used 3,072 TPU cores and 50,400 preemptible CPU cores for 44 days [9]. Given this limitation, there has been a need to create simplified, research-oriented environments that maintain the strategic and decision-making challenges of RTS games while reducing computational costs.

One of the most widely used examples is microRTS (μ RTS) [10]. Despite its minimalist design, μ RTS retains the same challenges as an RTS environment, such as unit control, resource management, and long-term planning. This simplicity has

¹Dune II: The Building of a Dynasty is based on David Lynch’s 1984 film Dune

²<https://www.ageofempires.com>

³<https://starcraft.blizzard.com>

made it a popular benchmark for evaluating algorithms such as Monte Carlo Tree Search.

Another important contribution is the ELF [11] platform, a research-oriented framework for Deep Reinforcement Learning (DRL) [12], which provides a lightweight, fast environment for large-scale experimentation without the computational barriers of large RTS games.

TorchCraft [13] uses a different approach. Instead of simplifying the game, it offers an interface that exposes the game state to learning algorithms. TorchCraft acts as a bridge between machine learning frameworks and StarCraft: Brood War, avoiding the computational cost of interacting directly with the game engine.

2.1 Planet Wars AI Challenge

Another example of a benchmark related to RTS games is the environment chosen for this project, which is Planet Wars, specifically the competition organized by the IEEE Conference on Games (CoG). The competition is held on a Fast Planet Wars platform optimized specifically for AI experimentation, capable of running extremely fast simulations (up to thousands of game ticks ⁴ per second), making it an ideal environment for large-scale agent testing under real-time constraints [6]. Unlike other resource-intensive RTS environments, such as StarCraft II, Planet Wars offers controllable complexity, reducing computational demands while maintaining significant strategic challenges.

The official competition repository ⁵ provides a highly modular and configurable system. The engine allows users to adjust key parameters such as the number of planets, fleet transit speed, growth rates, gravitational effects, and even the level of observability. Two standardized agent interfaces — PlanetWarsAgent (for fully observable games, see figure 2.1) and PartialObservationAgent (for limited-information environments) — enable researchers to test both complete and partial-knowledge scenarios. The repository also includes built-in competition modes, such as RoundRobinLeague, which facilitate the comparative evaluation of multiple agents under controlled conditions.

⁴Tick: It is a discrete unit of time used by the Planet Wars engine, responsible for processing the game logic, as well as serving as the game’s time counter.

⁵<https://github.com/SimonLucas/planet-wars-rts>

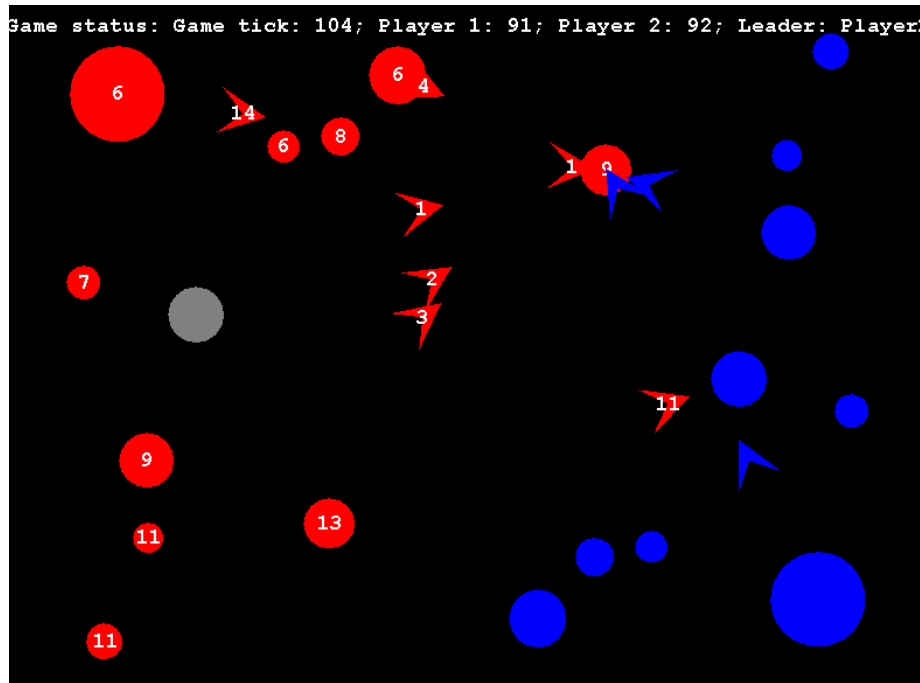


Figure 2.1: Screenshot of Planet Wars AI Challenge

2.2 Methods

The AI must deal with many uncertain conditions that change throughout the competition. Under these circumstances, the agent needs to adapt to the different environments that it will face. With all this in mind, previous participants and researchers have developed their own agents to deal with these uncertain conditions. In their own way, previous agents have 4 things in common: heuristic and simulation methods to calculate the best option at hand; evolutionary and optimization methods to learn from previous actions and improve strategies; reinforcement learning methods to autonomously acquire decision-making policies; and multi-agent systems and coordination to have multiple agents working in unison.

2.2.1 Heuristic and Simulation Methods

The earliest approaches to RTS control were based on custom heuristic rules, in which the agent evaluated possible actions based on parameters such as distance, planet growth rate, and enemy threat level [6]. The main drawback of this approach was

that, although these methods relied on interpretable logic and high computational efficiency, they often operated through scoring functions that evaluated and selected the most promising action frame by frame.

To overcome these limitations, forward simulation and rolling horizon planning techniques have been introduced. These methods simulate possible future states within a limited number of steps, allowing the agent to assess short-term consequences before committing to an action [3].

This approach was used by the winning team of the IEEE CoG 2025 competition, which combined a multi-factor heuristic evaluation with forward simulations to improve its strategic decision-making process.

2.2.2 Evolutionary and Optimization Methods

Evolutionary algorithms and metaheuristic optimization are methods frequently used in RTS games, such as Planet Wars. These methods have proven useful in scenarios with large, continuous action spaces. Given the limited time per action that the competition gives us, our agent needs to explore and refine strategies based on the current parameters and rules.

A notable example is the N-Tuple Bandit Evolutionary Algorithm (NTBEA), which combines evolutionary search with a probabilistic model to accelerate optimization in noisy and stochastic domains [14]. In an environment where scenarios and parameters are sensitive to change, such as Planet Wars, this algorithm helps reduce the amount of evaluation the agent performs, which is useful when time is limited.

Another bio-inspired approach involves immune-based optimization models that adjust agent behavior between attack and defense. By mimicking biological processes such as mutation, cloning, and selection, these algorithms improve agent adaptability in Planet Wars [15] or Ant Colony Optimization (ACO) [16]. This optimization method uses a pheromone system, similar to that of ants, to determine the best action to take; the less pheromone an action has, the lower the chance it is chosen. They help balance local and global search, allowing agents to adapt effectively to the changing conditions of the battlefield.

2.2.3 Reinforcement Learning Methods

Our agent must be capable of evaluating the best course of action, such as defending, attacking, or managing troops. To achieve this, developers use Reinforcement Learning (RL) [17], in which the agent tests different actions and receives rewards to learn policies that maximize the total reward of a given action [18] and, over time, generates a strategy. This method is a good approach for RTS games since one of their main characteristics is interaction with the environment, enabling the development of different strategies in response to the various rules our agent will face during the competition.

In Planet Wars and, in general, RTS games, multiple agents act individually; however, to reach a main goal, each action they take can affect the decisions of other agents, making teamwork indispensable for creating an effective strategy. Approaches like Multi-Agent Deep Deterministic Policy Gradient (MADDPG) [19] and QMIX [20] are good examples; these models apply a centralized training approach with a decentralized execution mechanism [21], where they share information from each agent during training but ultimately work with their own policies, thereby creating robust strategies.

2.2.4 Multi-agent Systems and Coordination

Multi-agent systems (MAS) provide a structure in which multiple autonomous entities cooperate or compete to achieve shared or conflicting objectives [22]. Each agent operates based on local decision-making, but the information it generates contributes to a global objective.

In Planet Wars, MAS serves as an effective decision-making tool by assigning individual agents to manage planet defense or to control attack fleets. Each agent focuses on a local goal while communicating with others to build a global information network. This method allows the system to remain operational even when one individual fails or behaves unpredictably.

However, MAS also presents challenges for RTS environments, where decisions must be made within a short time frame. Communication between agents and the coherence of shared information can be very challenging. If communication becomes overloaded or desynchronized, system performance can be severely affected, leading to longer evaluation periods.

3. Methodology

In this chapter, we describe the design, development, and evaluation of our intelligent agent, present the research paradigm of our project, and outline the steps to follow during the iterative design and experimentation cycles. In this way, the aim is to establish a conceptual and procedural foundation for the work.

Planet Wars is a game that involves capturing planets and fighting opponents, all with the same objective: to gain control of the map. Every player starts with the same number of planets, which gradually generate ships to later be sent to other planets, whether neutral or already belonging to a rival. These ships do not fight each other; they travel across the map to their target and capture it. A planet can pass into a player’s possession by eliminating the enemy troops that are already on that planet. For example, if a planet has 15 ships at the moment, and a fleet of 20 ships is sent, the planet passes to our possession with a total of 5 ships in our favor. This game has 2 ways to end a match: either time runs out, and the winner is the one with the most planets in possession, or all of the enemy rival’s planets are captured. In both cases, it may be that some planets remain neutral; however, this does not affect the result of the game. For this version of the game, complex user interfaces are not implemented; Planet Wars is a simplified but highly strategic environment.

We aim to develop a competitive AI agent for the Planet Wars AI Challenge, which is, in essence, a computational artifact. The proposed methodology consists of designing and developing the agent, and then evaluating its performance in the dynamic environment of Planet Wars. This approach falls within the Design Science Research (DSR) [23] paradigm, a combined method used for research and software engineering projects.

3.1 Research Paradigm

DSR focuses on solving problems by creating artifacts and generating knowledge from the design process. This paradigm is organized into an iterative cycle composed of the stages of Problem Awareness, Suggestion, Development, Evaluation, and Conclusion [24]. In the context of this project, the artifact corresponds to the intelligent agent, whose construction requires iterating between design, implementation, and evaluation through experiments in the Planet Wars environment.

Using this methodology is ideal for guiding the development of our intelligent agent since DSR is widely used in research on software and computational artifacts, as is the case with our competitive AI. One of the reasons supporting this decision is how we divided this project: designing and developing a competitive agent, evaluating it in the competition’s official repository, and generating results from the strategies used to determine the most effective approach.

DSR has previously been used in artificial intelligence projects, as it allows the design process of models, agents, or decision techniques to be structured systematically and incorporates experimental evaluations within the development framework to improve their performance [25].

3.2 Design Science Research Framework

In this section, we will explain the procedure to be carried out in the 5 steps outlined by DSR. The main idea is to plan how to execute and evaluate this project and to identify potential results that could improve the performance of our AI agent.

- **Awareness of Problem:** We know that Planet Wars is a dynamic environment that changes throughout the competition; elements such as rules, parameters, and maps are examples of this. In this changing environment, the main challenge is to design an AI agent that can adapt to these changes, make reliable decisions, and develop robust strategies.
- **Suggestion:** Based on agents who participated in previous competitions and on prior research, four methodologies used for RTS environments stand out as the most suitable for Planet Wars: heuristic models, evolutionary search, reinforcement learning, and multi-agent coordination. The project proposes a

modular architecture that enables the testing and comparison of the different approaches mentioned to identify a strong candidate for the Planet Wars AI Challenge.

- **Development:** The development is based on an iterative testing cycle [26], where the agent tests different approaches independently. The first of these is testing heuristic methods like Monte Carlo Tree Search (MCTS) [27], which have shown good results in RTS environments. However, this isn't limited to just one approach; the iterative cycle allows us to revisit our steps and test other approaches. The procedure is the same for the remaining methods. While their development is carried out independently, one objective is to control multiple agents. Considering methods like NTBEA as reinforcement learning algorithms, a hybrid agent is feasible. This part of DSR allows us to create and test multiple methods, enabling us to find an effective strategy.
- **Evaluation:** The agent evaluation takes place in the official competition repository, which provides random and heuristic agents as baselines for competing against ours. Also, we are going to use the results of the competition as metrics evaluators ¹. This environment, designed for AI research, allows adjustments to parameters such as decision time, number of planets, transition speed, and more; it enables multiple scenarios to test our agent. Win rates are the primary metric for evaluating our agent.
- **Conclusion:** With the results obtained, we can identify the strengths and limitations of each approach, determining which methods strike a balance between agent performance and computational cost across the games played.

3.3 Experimental Design

The experimental design establishes the characteristics of the simulation environment, the baseline agents we will use to test our agents, and the metrics we will use to evaluate their performance and efficiency in Planet Wars.

¹<https://github.com/SimonLucas/planet-wars-rts/tree/main/competitions>

3.3.1 Environment Configuration

The official repository for the competition provides two interfaces for testing our agent: **PlanetWarsAgent**, which offers a visualization of our agent’s performance, showing planets, troop movements, and the game’s progress; and **PartialObservationAgent**, which, as its name suggests, provides partial information about the environment and does not show the game’s progress. Both interfaces can be used to evaluate our agent and for AI-related research. For this project, we have chosen to use the full version of the environment.

Regardless of the visualization mode used at any given time, various scenarios are employed to evaluate the agent’s adaptability across different environments. Various parameters will change, generating everything from small, balanced maps to asymmetrical, high-pressure situations. These parameters are:

- Number of planets and their spatial distribution.
- Planet growth rates.
- Fleet travel speed.
- Map size and gravitational modifiers.
- Time available per decision step.
- Number of initial ships and neutral planets.

3.3.2 Baseline Agents

Our agent will be compared with other benchmark agents from the repository to assess performance. Each of these adversaries represents a different level of complexity to which our AI is subjected, allowing us to measure improvements in adaptability and the strategies employed. Some of these agents include **RandomAgent**, which acts randomly rather than strategically, thereby reducing complexity. It is also compared with **heuristic agents** that follow predefined rules and prioritize key factors such as distance to valuable resources and potential threats.

Other agents that aid in the evaluation of our work are those designed in previous studies, notably those from TeamTitans, NanaBoshi, and GalacticArmada ². These

²https://github.com/SimonLucas/planet-wars-rts/blob/main/competitions/IEEE_CoG_2025_Results.md

agents utilize evolutionary methods or basic planning, delivering strategies that exploit exploration with less adaptability. Together with the agents available in the repository, these agents form a progression of difficulty from minimal competence to a realistic challenge.

3.3.3 Metrics

The win rate will be the primary metric for evaluating and comparing our agent's performance across the different scenarios tested. While it is a simple metric, it is the standard measuring system frequently used in intelligent agent competitions and research [28], facilitating the evaluation and comparison of our agent and its strategy against the rival agent.

However, our agent's performance cannot be measured solely by their win rate. Efficient behavior is a key factor in evaluating performance in real-time strategy environments [29]. To provide a more comprehensive assessment, we will also measure planet control, fleet control, score difference, and decision time. In this way, we focus not only on the agent's win-loss rate but also on the quality of their decisions and resource management, identifying potential weaknesses. The values used for these metrics are obtained after 50 games per experimental condition against other agents from the official repository.

3.3.4 Implementation Plan

The next figures 3.1 3.2 show the Gantt diagrams for this project, integrating the state of the art research and agent design architecture carried out this year and the next steps to be taken.

Gantt Diagram - Intelligent Agent for Planet Wars 2025

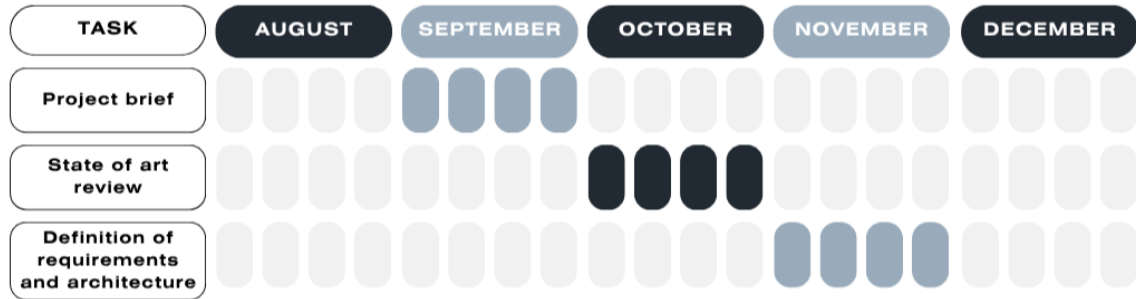


Figure 3.1: Gantt Diagram 2025

Gantt Diagram - Intelligent Agent for Planet Wars 2026

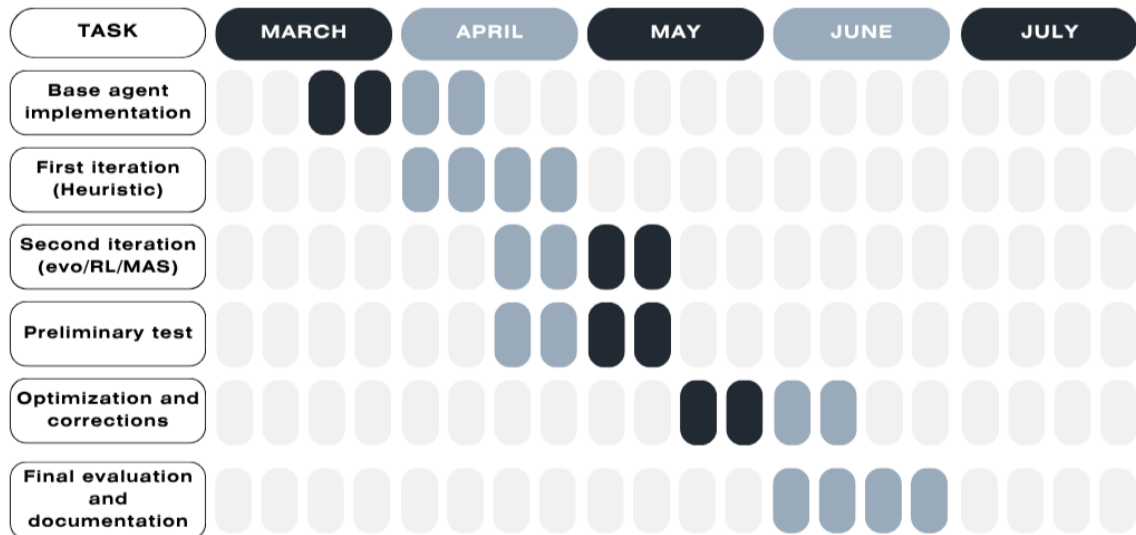


Figure 3.2: Gantt Diagram 2026

4. Development of the ISILab Agent

The Planet Wars code supports two programming languages for creating our agent: Python and Kotlin. Python is a high-level language that offers a wide range of useful libraries for developing AI software in the RTS environment. For example, the PyTorch library [30] and TensorFlow [31] are open-source machine learning frameworks. These two libraries are fully supported by the Planet Wars competition.

Kotlin is also a high-level programming language supported by the Planet Wars competition; in this case, the repository offers full access to the forward model, allowing work with the MCTS algorithm. Given that execution is one of the most important factors in the competition, Kotlin surpasses Python because it is a compiled language that runs on the Java Virtual Machine (JVM) [32], enabling the rapid execution of large numbers of simulations. For this reason, Kotlin was selected for developing the ISILab Agent.

The agent’s architecture is based on the best-first search method, Monte Carlo Tree Search (MCTS) [33], which is adapted for the Planet Wars environment. The agent can determine whether it is at an advantage or disadvantage and prioritize exploration in the early game. This allows the agent to explore multiple actions, even if their stats aren’t the best yet. As the game progresses, this uncertainty decreases, and the agent already knows which actions are best to execute. By the late game, the system shifts its focus to exploiting the actions that have shown the best results against the rival agent.

4.1 Architecture

The ISILab Agent works with 2 main classes: MCTS, which is responsible for real-time decision-making, and a GameState Wrapper that translates engine information to enable MCTS.

4.1.1 Monte Carlo Tree Search

As previously mentioned, the competition engine greatly facilitates the implementation of MCTS. This search tree works by repeatedly executing four distinct phases: Selection, Expansion, Simulation, and Backpropagation. In this figure 4.1, we show the standard diagram illustrating how MCTS works.

Selection

The first phase of MCTS is selection. This phase starts at the root node and descends through the tree, selecting the most promising child node at each level until it reaches a node with unexplored actions. This is where UCT is applied. The exploration constant varies according to the progress of the game, which is measured based on the proportion of ticks elapsed relative to the maximum limit of 200 iterations, as well as the estimated advantage given by the difference in ship count and the growth rate of both players. Unlike traditional MCTS, this one integrates a Progressive Bias, a heuristic evaluation that helps to explore during the early phases of the match, guiding the agent towards promising actions while reducing the branching factor. As more nodes are visited, this bias decreases to the point where it is no longer necessary to use it, since enough simulations have been performed by this point in the game for UCT to take control of the selection.

Expansion

When a node with unexplored actions is reached, a new child node is created, and one of the available actions from the current state is executed. In this phase, the agent clones the current game state using `deepCopy()`, simulating the game physics, including fleet movement, planetary growth, and combat resolution.

The available actions for the tree are strategically scored before expansion during the early game to work with the progressive bias. This ensures that the tree grows

preferentially toward relevant states rather than exploring arbitrary moves at the beginning of the match.

Simulation

Once a new node has been expanded, the simulation phase begins, also known as the rollout or playout. The Planet Wars engine provides `ForwardModel()`, allowing the agent to try actions many ticks forward. From the newly created state, the agent applies actions at random. However, this rollout is not purely random; it is guided by progressive bias during the early game. This process is repeated until the state becomes terminal or the maximum rollout depth is reached.

At the end of the simulation, the resulting state is evaluated using a heuristic function that measures differences in ship count, growth rate, ships in transit, and territorial control. These factors are weighted based on the state of the game. When a terminal state is reached, a fixed reward of +100 is returned if the player is winning or -100 if the player is losing, on the same heuristic scale.

Backpropagation

Once the simulation returns a score, the backpropagation phase updates the statistics of every node visited during that iteration, propagating from the newly created leaf node back up to the root of the tree. In each iteration, the visited nodes increment their visit counter by 1 and accumulate the result obtained from the simulation. The final action selection is not performed using the UCT formula. Instead, the agent selects the child node with the most visits.

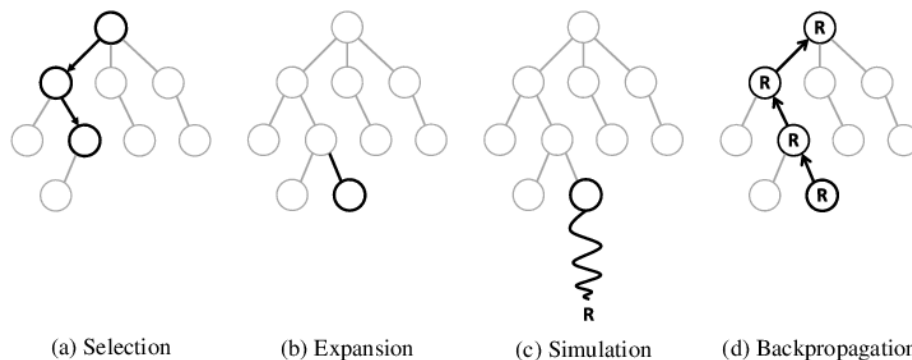


Figure 4.1: Monte Carlo Tree Search [34].

For this MCTS implementation, the Upper Confidence Bound applied to Trees (UCT) [35] is modified. Instead of using the standard equation 4.1, the constant (C) was turned into a variable, allowing a dynamic transition from exploration to exploitation based on the progress of the game. Additionally, to support this adaptability, a progressive bias [36] was incorporated. This bias uses heuristic evaluators to aid exploration in the early stages, helping to identify the best actions. The formula for the bias 4.2 makes it gradually decrease as more nodes are visited, eventually becoming unused in the late game.

$$UCT = \frac{Q_i}{n_i} + C\sqrt{\frac{\ln N}{n_i}} \quad (4.1)$$

$$PB_i = \frac{H_i}{1 + n_i} \quad (4.2)$$

- Q_i = total sum of rewards (wins) for node i
- n_i = number of visits to node i
- N = total number of visits to the parent node
- C = exploration constant
- H_i = heuristic evaluation of node i

The complete formula used by the agent is:

$$UCT_i = \frac{w_i}{n_i} + C_v\sqrt{\frac{\ln N}{n_i}} + PB_i \quad (4.3)$$

4.1.2 GameState Wrapper

This module is responsible for connecting MCTS with the Planet Wars environment. For the tree to work, it requires an interface called MCTSState, which defines the following operations:

1. `isTerminal()`: Determine whether the iteration has reached the maximum limit or if there are no more opponents' planets to conquer.
2. `getResult()`: Returns the evaluation of the state.

3. `takeAction()`: Applies an action and returns the new state.
4. `takeRandomAction()`: Executes a random action.
5. `getActions()`: Generates the available actions.
6. `stateQuickEval()`: Fast heuristic evaluation.

It is here that our wrapper acts as a translator, enabling all operations provided by the game engine to interact with the `MCTSState` interface. This allows the search tree to understand the game domain and progress.

Action generator

To generate the action of the agent, the method `getActions(playerId: Player)` performs that role. This method uses priority logic to generate defensive and offensive actions, calculated with lightweight heuristics. The main objective of this method is to minimize the branching factor of MCTS and guide the search toward satisfactory strategic moves.

- **Defensive Actions:** The agent evaluates its own planets that are in danger by detecting the enemy fleets moving towards them. For each planet in danger, reinforcements are generated from allied planets near them, calculating the number of ships necessary to nullify the threat.
- **Offensive Actions:** Each available planet in possession evaluates the possible objectives using a heuristic calculation that considers the viability of the attack, the redundancy of attacks, the temporal urgency, and the growth rate of the target planet. This function prioritizes enemy planets over neutral planets; once the best objectives are selected, the number of ships to attack is calculated.

This method includes the method `Action.doNothing()`, which is provided by the Planet Wars engine. These 3 actions are shuffled to avoid order-based bias.

State evaluation

The agent uses 2 methods to evaluate states: one with low computational cost and the other for complete evaluation.

- `getResult(playerId: Player)`: This method returns a complete evaluation of the agent’s state, calculating the difference between total ships, territorial control, and growth potential. Some of these elements are calculated with dynamic weights based on the match’s progress, such as `earlyGrowthWeight`, `lateGrowthWeight`, `earlyTransitWeight`, and `lateTransitWeight`. This allows the agent to prioritize growth at the beginning of the match and territorial control at the end. Once a state becomes terminal, it delivers extreme rewards according to its superiority over the rival.
- `stateQuickEval(playerId: Player)`: Unlike the previous method, this one returns a lightweight heuristic evaluation based on the difference in ships and the growth rate. This function is used during the selection phase of MCTS to compute a lead factor and adjust the exploration constant.

4.2 Hyperparameter optimization

As we progressed in developing the ISILab Agent, many parameters began to appear as more actions or complementary functions we created; however, the value of each parameter was assigned manually to test functionality against expected behavior. Once the agent works correctly, we employ Hyperparameter Optimization (HPO) to identify the most suitable value for each hyperparameter using the Irace Package [37] from R software [38].

Irace is an automatic algorithm configuration tool based on Iterated F-Race analysis [39], seeking to deliver possible candidates with promising values through evaluations within the context of the problem; in this case, the Planet Wars matches. This is an iterative process that statistically eliminates the worst values until it converges to 5 possible cases.

The agent has 14 hyperparameters distributed across behavioral actions and evaluation functions.

- **Behavioral Actions:** These parameters determine the thresholds and fractions that govern offensive and defensive decision-making. They do not decide which action to take, but rather evaluate the minimum proportion of ships necessary for a possible attack, including both the fraction of ships to deploy for an

attack and the minimum number of ships that must remain on the planet before sending reinforcements without being undefended.

- **Evaluation Functions:** These functions determine which planets are conquerable or require defense. Different weights are used in the calculations of these functions; they account for the growth rate of troops, the speed of ships, the terrain covered, and the priority of planets (enemy, own, neutral).

Irace calls an external evaluator that runs real Planet Wars matches using the competition's reference agents: *CarefulRandomAgent* and *GreedyHeuristicAgent*. Each configuration performs 50 matches, alternating the agent's position (player 1, player 2) to eliminate positional bias. The performance of each configuration is defined as $1 - \text{winRate}$; the goal of Irace is to decrease this value, which means increasing the winRate.

4.3 Versions of the ISILab agent

Following the iterative process of DSR, three versions of the ISILab agent were developed:

1. **Version 0:** This version of the agent serves as the basis for the following versions. Here, MCTS is implemented for a multi-bandit system integrated into UCT, and is modified by replacing its constant with a dynamic one to prioritize exploration at the beginning and later exploit actions at the end of the match. However, this version was not submitted to the competition because it was very basic and its parameters were hand-tuned.
2. **Version 1:** This version of the agent was submitted for the competition; here, hyperparameter optimization is already implemented to improve performance with respect to the values of the variables used in decision-making. However, this version is excluded from the competition due to its low effectiveness against the Greedy Heuristic agent.
3. **Version 2:** The last version developed, a heuristic method known as progressive bias, to guide the initial exploration in the game and later decrease its influence in decision-making. This implemented method shows a significant

improvement in the agent's performance. This version is accepted in the competition, showing remarkable performance against the Greedy Heuristic Agent and Careful Random Agent, the main evaluators of the competition, and is surpassed only by agents developed by other participants. This version mixes multiple agents, hyperparameter optimization, and heuristic methods.

5. Evaluation and results

In this chapter, we evaluate the performance of the ISILab Agent using the `EvaluateAgent.kt` runner class. This runner simulates matches where the agent plays on each side of the map n times, as defined by us. If $n = 3$, then `EvaluateAgent` runs 6 matches.

The engine of Planet Wars provides 3 types of agents that we can use to play against the ISILab Agent.

- Random agents: Pure Random Agent, Better Random Agent, Careful Random Agent, Slow Random Agent, and Heavy Computation Agent.
- `EvoAgent-200-20-0.5-true` (Evolutive agent).
- Greedy Heuristic Agent.

Also, we're going to use the results of the competition of the Planet Wars AI Challenge carried out in the International Conference on Autonomous Agents and Multiagent Systems 2026 (AAMAS) ¹, where ISILab Agent faced agents created by other participants.

5.1 Experimental Setup

`EvaluateAgent` runs matches using the real parameters used during the competition to evaluate our agent's performance and determine whether the changes we are implementing improve the ISILab Agent.

¹<https://cyprusconferences.org/aamas2026/competitions/>

For our evaluation, the agent plays 25 matches for each side, for a total of 50 matches; these are played within predefined parameter bounds. Each match randomly selects the values, forcing the agents to adapt to scenarios that change throughout the competition. The parameters defined by the official competition repository ² are:

- Number of planets: 10 – 30.
- Neutral ratio: 0.25 – 0.35.
- Growth rate: 0.05 – 0.2.
- Transporter speed: 2.0 – 5.0 units/tick.
- Game duration: 2000 ticks.

According to the competition, games will run at 20 Hz, giving agents 50 ms per decision. However, this is a real-time competition: agents may take longer (but will act on stale information), and if both respond quickly, the game will run faster.

The results obtained after these evaluations are calculated using the agent’s win rate as a reference. This metric is commonly used to evaluate the performance of an AI in RTS game environments [40]. It is also the metric used for the official competition, which is why we use it to evaluate our agent.

5.2 Results

The results obtained belong to the 3 versions of the ISILab agent. The first table 5.1 presents the win rates of the different versions, our main measure, while the following table 5.2 provides relevant computational performance metrics such as average action time and total timeouts. Each version plays 50 games against the agents in the repository³.

²https://github.com/SimonLucas/planet-wars-rts/blob/main/competitions/AAMAS_2026.md

³In this context timeouts means the agent returns the action `DoNothing()`

Table 5.1: Win rate of ISILab Agent versions against repository agents

Opponent	V0	V1	V2
Pure Random Agent	98.0%	100.0%	100.0%
Better Random Agent	98.0%	96.0%	100.0%
Careful Random Agent	96.0%	76.0%	100.0%
Heavy Computation Agent	94.0%	100.0%	100.0%
Slow Random Agent (delayMillis: 1000ms)	100.0%	100.0%	100.0%
EvoAgent-200-20-0.5-true	72.0%	62.0%	100.0%
Greedy Heuristic Agent	12.0%	68.0%	94.0%
Overall Average	81.4%	86.0%	99.1%

Table 5.2: Performance metrics of ISILab Agent versions

Metric	V0	V1	V2
Average action time (ms)	54.54	86.0	55.42
Total timeouts	42.685	11.082	76.999

The results of version 0 of the agent show good performance against random agents, with a victory rate above 94%, demonstrating that it has strategic behavior; however, against agents with strategic behaviors such as EvoAgent and the greedy agent, the victory rate drops noticeably, especially with the greedy agent reaching only a 12% win rate. The agent’s average action time is 54.54 ms, slightly exceeding the competition’s proposed limit. The agent has a total of 42,685 timeouts throughout the 350 games in which it returns DoNothing().

Version 1 of the agent, the first one submitted to the competition, has hyperparameter optimization implemented. The results show that the victory rate against the greedy agent increases notably from 12% to 68%; however, the performance against random agents and the EvoAgent decreases. Although there is an improvement over the previous version, its performance is not yet satisfactory for the competition. The average action time far exceeds the 50 ms margin; however, it has shorter waiting times than the other 2 versions, with DoNothing() occurring only 11 times.

Version 2 of the agent integrates the progressive bias, reaching its best performance. The victory rate increases to 100% against random and evolutionary agents, with only the greedy agent posing a non-significant challenge to the agent. The agent returns to an average margin of 55.42 ms per action; however, timeouts increased to 77, during which the agent returns DoNothing() in 350 games.

In addition to evaluating against the agents in the repository, an evaluation was conducted among the 3 versions of the ISILab agent. Each version played 100 games against each version of the agent. The data presented in the table shows the progressive improvement of this agent.

Table 5.3: Win rate between ISILab Agent versions (100 games per matchup)

Agent	vs V0	vs V1	vs V2	Overall Average
V0	-	2.0%	5.0%	3.5%
V1	94.0%	-	15.0%	54.5%
V2	100.0%	95.0%	-	97.5%

- **V0:** Average action time = 38,92 ms, Total timeouts = 324
- **V1:** Average action time = 35,41 ms, Total timeouts = 335
- **V2:** Average action time = 37,54 ms, Total timeouts = 197

Version 0 had the lowest performance against its later versions, having an average of 3.5% victories. Version 1, which includes hyperparameter optimization, improves its performance with an average of 54.5% victories, resulting from its superiority over the previous version. The last version of the agent that integrates progressive bias has been clearly superior, with an average of 97.5% victories. These results show the effectiveness of the improvements introduced in each iteration of this project.

By demonstrating favorable performance, especially against the GreedyHeuristic and CarefulRandom agents, the competition's main evaluators. The second version of the agent is accepted to participate against other developers' agents. In the Planet Wars AI Challenge ⁴, multiple agents can be submitted by the same participant; the

⁴<https://github.com/SimonLucas/planet-wars-rtts-submissions/blob/be28485933272e1109c65d5ec705bbd914c977b4/results/spring-2026/leaderboard.md>

following table 5.4 summarizes the competitors and the positions in which each finished.

Table 5.4: Summary of the best version of the agent from the competition

Rank	Agent	μ	σ	$\mu - 3\sigma$
1	teamtitanagentv3	33.600	0.737	31.388
2	gnn_cont999_128_top64	32.503	0.676	30.475
3	nanaboshi08271	29.064	0.605	27.249
4	adversarial-dba	24.965	0.515	23.420
5	galacticarmada	22.758	0.489	21.290
6	ISILab-agent	20.852	0.491	19.380
7	conditional-greedy-agent	19.682	0.471	18.269
8	frontiersentryhyperguardplus	18.420	0.467	17.019
9	bastiontempov4_tunablestrategic	17.260	0.467	15.857
10	ease_agent_001	15.935	0.483	14.486
11	intelligentagent	15.386	0.475	13.962
12	adaptivefrontiersentinel	14.359	0.484	12.906

The competition is classified using the metric $\mu - 3\sigma$. The repository assigns its own score, where μ represents the agent’s average performance across the games played, and σ is the standard deviation, which measures per-game stability, . This equation penalizes agents with high performance variability and favors consistent, reliable behavior.

Our agent, ISILab-agent, obtained an average score of $\mu = 20.852$ with a standard deviation of $\sigma = 0.491$. The lower limit at $\mu - 3\sigma$ reached 19.380 points. Assuming a normal distribution of the results, approximately 99.7% of the agent’s executions should obtain a score higher than this value, which demonstrates a good level of performance and acceptable stability. This result is particularly satisfactory, since it is the agent’s first participation in the competition against established opponents such as TeamTitans, NanaBoshi, and GalacticArmada.

These results support the effectiveness of the improvements implemented throughout the agent’s development, especially the progressive bias, which enables the agent to achieve near-perfect victory rates. The actual results of the competition can be

found in the official repository for the Planet Wars AI Challenge.

5.3 Evaluation

The Fisher exact test [41] is used because the data obtained is binary; they measure the agent’s victory or defeat. In addition, the scenarios in which the agent is tested are not similar but independent, with variations in the number of planets, troop movements, and even the agent’s starting position, since each map is generated randomly.

Fisher proposes 2 hypotheses: null (H_0) and alternative (H_1). H_0 proposes that there is no change or improvement in the performance of the agent, indicating that the probability of the ISILab agent winning is the same, regardless of the version used. On the other hand, H_1 establishes that the victory rate of some versions of the agent differs significantly from that of others, indicating that iterative development has produced a measurable improvement. For this evaluation, the data to be measured are the overall win-rate results using a decision threshold of $\alpha = 0.05$. If the p-values are less than α , the implemented changes show a significant improvement; if they are greater than this threshold, H_0 is not rejected.

Table 5.5: Fisher’s Exact Test – Repository agents

The Odds Ratio in this context represents the relative odds of winning.

Comparison	Win Rate	p-value	Odds Ratio
V0 vs V1	81.4% vs 86.0%	0.124	1.4
V1 vs V2	86.0% vs 99.1%	< 0.001	18.8
V0 vs V2	81.4% vs 99.1%	< 0.001	26.3

Table 5.6: Fisher’s Exact Test – Versions

The Odds Ratio in this context represents the relative odds of winning.

Comparison	Win Rate	p-value	Odds Ratio
V0 vs V1	3.5% vs 54.5%	< 0.001	32.7
V1 vs V2	54.5% vs 97.5%	< 0.001	32.3
V0 vs V2	3.5% vs 97.5%	< 0.001	999.0

The first table 5.5 presents the performance results of the 3 versions against the official agents from the repository. Although there is an increase in the win rate for each version, H_0 is not discarded; the first comparison, between V0 and V1, gives a p-value = 0.124, which is higher than the defined threshold. Without a doubt, it is worth highlighting that it improves notably over the Greedy Heuristic Agent, going from 12% to 68% in win rate, but the overall average increases slightly, indicating a non-significant improvement. However, the results change when evaluating V1 against V2, and especially V0 against V2; in these cases, H_1 is validated, with a p-value below the threshold, demonstrating that V2 has 19 times higher odds of victory against V1 and 26 times the chance against V0. This demonstrates that the improvements progressively enhance the agent’s performance.

The second table 5.6 presents results from matches between versions of the ISILab agent, excluding results against external agents. In this evaluation, H_0 is discarded in its entirety, in contrast to the first table, where V0 and V1 do not show a significant improvement. These results confirm the agent’s progressive improvement, demonstrating that implementing hyperparameter optimization for V1 and introducing progressive bias in V2 improve the agent’s performance, with V1 having 33 times more chances of victory against V0 and V2 having 32 times more chances of victory.

6. Conclusion

This work presented the development of an intelligent agent capable of competing in the Planet Wars AI Challenge, fulfilling the proposed general objective. The architecture of the ISILab agent combines 3 of the 4 methods presented in the state of the art through the iterative process proposed by DSR, utilizing the multiple-bandit system of UCT that integrates MCTS, the hyperparameter optimization of IRace, and the heuristic calculation of progressive bias.

The results from the competition show that the combination of these 3 methods, frequently used in RTS games, achieves competitive performance in the dynamic environments of Planet Wars, as evidenced by the agent’s win rate. In particular, the implementation of progressive bias, together with the dynamism of the exploration constant, was key to both exploration and exploitation in MCTS.

Obtaining a score of $\mu - 3\sigma = 19.380$, with an average of $\mu = 20.852$ and a standard deviation of $\sigma = 0.491$, the ISILab agent ranks 17th out of 30 participants in the competition, validating the competitive and strategic behavior of the intelligent agent. This result is considered acceptable given that it is the first time participating in this competition.

However, the results obtained outside the competition, in the official repository evaluator, demonstrate important limitations for future improvements of the agent, such as the speed of decision-making, which slightly exceeds the 50 ms that Planet Wars gives, causing the agent to tend to deliver `DoNothing()`, allowing other agents whose decision-making speed exceeds the performance of the ISILab agent. Therefore, despite the competitive performance achieved, future work should focus on improving decision-making speed.

This project can serve as a foundation for future participation in AI competitions and for research on decision-making in RTS games.

Bibliography

- [1] Raúl Lara-Cabrera, Carlos Cotta, and Antonio J Fernández-Leiva. A review of computational intelligence in rts games. In *2013 IEEE Symposium on Foundations of Computational Intelligence (FOCI)*, pages 114–121. IEEE, 2013.
- [2] Zane Durante, Qiuyuan Huang, Naoki Wake, Ran Gong, Jae Sung Park, Bidipta Sarkar, Rohan Taori, Yusuke Noda, Demetri Terzopoulos, Yejin Choi, et al. Agent ai: Surveying the horizons of multimodal interaction. *arXiv preprint arXiv:2401.03568*, 2024.
- [3] Santiago Ontanón. Combinatorial multi-armed bandits for real-time strategy games. *Journal of Artificial Intelligence Research*, 58:665–702, 2017.
- [4] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *nature*, 575(7782):350–354, 2019.
- [5] Chengpeng Hu, Yunlong Zhao, Ziqi Wang, Haocheng Du, and Jialin Liu. Games for artificial intelligence research: A review and perspectives. *IEEE Transactions on Artificial Intelligence*, 5(12):5949–5968, 2024.
- [6] Simon M Lucas. Game ai research with fast planet wars variants. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–4. IEEE, 2018.
- [7] Antonio Maciá-Lillo, Antonio Jimeno-Morenilla, Higinio Mora, and Eduard Duta. Hybrid architecture for ai-based rts games. *IEEE Transactions on Games*, 2025.

- [8] Oriol Vinyals, Igor Babuschkin, Junyoung Chung, Michael Mathieu, Max Jaderberg, Wojciech M Czarnecki, Andrew Dudzik, Aja Huang, Petko Georgiev, Richard Powell, et al. Alphastar: Mastering the real-time strategy game starcraft ii. *DeepMind blog*, 2:20, 2019.
- [9] Shengyi Huang, Santiago Ontañón, Chris Bamford, and Lukasz Grela. Gym- μ rts: Toward affordable full game real-time strategy games research with deep reinforcement learning. In *2021 IEEE Conference on Games (CoG)*, pages 1–8. IEEE, 2021.
- [10] Santiago Ontañón, Nicolas A Barriga, Cleyton R Silva, Rubens O Moraes, and Levi HS Lelis. The first microrts artificial intelligence competition. *AI Magazine*, 39(1):75–83, 2018.
- [11] Yuandong Tian, Qucheng Gong, Wenling Shang, Yuxin Wu, and C Lawrence Zitnick. Elf: An extensive, lightweight and flexible research platform for real-time strategy games. *Advances in Neural Information Processing Systems*, 30, 2017.
- [12] Per-Arne Andersen, Morten Goodwin, and Ole-Christoffer Granmo. Deep rts: a game environment for deep reinforcement learning in real-time strategy games. In *2018 IEEE conference on computational intelligence and games (CIG)*, pages 1–8. IEEE, 2018.
- [13] Gabriel Synnaeve, Nantas Nardelli, Alex Auvolat, Soumith Chintala, Timothée Lacroix, Zeming Lin, Florian Richoux, and Nicolas Usunier. Torchcraft: a library for machine learning research on real-time strategy games. *arXiv preprint arXiv:1611.00625*, 2016.
- [14] Simon M Lucas, Jialin Liu, and Diego Perez-Liebana. The n-tuple bandit evolutionary algorithm for game agent optimisation. In *2018 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–9. IEEE, 2018.
- [15] Mohamed Amine Chikh Touami, Mohammed SALEM, and Mohammed Faycal KHELFI. Optimizing planet wars bot using an immune-based algorithm. In *2018 International Conference on Applied Smart Systems (ICASS)*, pages 1–7. IEEE, 2018.
- [16] Alejandro Baldominos, M González-Evstrópova, Emilio Martín, and Yago Sáez. Planet wars: an approach using ant colony optimization. In *Proceedings of the In-*

ternational Conference on Metaheuristics and Nature Inspired Computing (META), oct 2018.

- [17] Zhengyang Li, Qijin Ji, Xinghong Ling, and Quan Liu. A comprehensive review of multi-agent reinforcement learning in video games. *IEEE Transactions on Games*, 2025.
- [18] Nesma M Ashraf, Reham R Mostafa, Rasha H Sakr, and MZ Rashad. A state-of-the-art review of deep reinforcement learning techniques for real-time strategy games. *Applications of Artificial Intelligence in Business, Education and Healthcare*, pages 285–307, 2021.
- [19] Ryan Lowe, Yi I Wu, Aviv Tamar, Jean Harb, OpenAI Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. *Advances in neural information processing systems*, 30, 2017.
- [20] Tabish Rashid, Mikayel Samvelyan, Christian Schroeder de Witt, Gregory Farquhar, Jakob Foerster, and Shimon Whiteson. Monotonic value function factorisation for deep multi-agent reinforcement learning. *Journal of Machine Learning Research*, 21(178):1–51, 2020.
- [21] Weigui Jair Zhou, Budhitama Subagdja, Ah-Hwee Tan, and Darren Wee-Sze Ong. Hierarchical control of multi-agent reinforcement learning team in real-time strategy (rts) games. *Expert Systems with Applications*, 186:115707, 2021.
- [22] Davide Calvaresi, Yashin Dicente Cid, Mauro Marinoni, Aldo Franco Dragoni, Amro Najjar, and Michael Schumacher. Real-time multi-agent systems: rationality, formal model, and empirical results. *Autonomous agents and multi-agent systems*, 35(1):12, 2021.
- [23] Jan Vom Brocke, Alan Hevner, and Alexander Maedche. Introduction to design science research. In *Design science research. Cases*, pages 1–13. Springer, 2020.
- [24] Alan Hevner and Samir Chatterjee. *Design Research in Information Systems: Theory and Practice*. Springer Publishing Company, Incorporated, 1st edition, 2010.
- [25] Michael Gau, Felix Kretzer, Alexander Maedche, and Jan vom Brocke. Ai-based design science research: An exploratory framework for leveraging artificial intelligence

- in design science research. In *International Conference on Design Science Research in Information Systems and Technology*, pages 18–31. Springer, 2025.
- [26] Marlien Herselman and Adele Botha. Evaluating an artifact in design science research. In *Proceedings of the 2015 Annual Research Conference on South African Institute of Computer Scientists and Information Technologists*, pages 1–10, 2015.
- [27] Zuozhi Yang and Santiago Ontanón. Guiding monte carlo tree search by scripts in real-time strategy games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 15, pages 100–106, 2019.
- [28] Hyeon-Chang Jeon, In-Chang Baek, Cheong-mok Bae, Taehwa Park, Wonsang You, Taegwan Ha, Hoyoun Jung, Jinha Noh, Seungwon Oh, and Kyung-Joong Kim. Raidenv: Exploring new challenges in automated content balancing for boss raid games. *IEEE Transactions on Games*, 16(3):645–658, 2023.
- [29] Man-Je Kim, Kyung-Joong Kim, Seungjun Kim, and Anind K Dey. Performance evaluation gaps in a real-time strategy game between human and artificial intelligence players. *IEEE Access*, 6:13575–13586, 2018.
- [30] Sagar Imambi, Kolla Bhanu Prakash, and GR Kanagachidambaresan. Pytorch. In *Programming with TensorFlow: solution for edge computing applications*, pages 87–104. Springer, 2021.
- [31] TensorFlow Developers. Tensorflow. *Zenodo*.
- [32] Santhosh Chitraju Gopal Varma. Comparing java with python and kotlin: A comprehensive analysis of performance, syntax, and usability. *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, 2(2):30–38, 2021.
- [33] Mark HM Winands. Monte-carlo tree search. In *Encyclopedia of computer graphics and games*, pages 1179–1184. Springer, 2024.
- [34] Steven James, George Konidaris, and Benjamin Rosman. An analysis of monte carlo tree search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.
- [35] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

- [36] Guillaume Maurice Jean-Bernard Chaslot Chaslot. Monte-carlo tree search. 2010.
- [37] Manuel López-Ibáñez, Leslie Pérez Cáceres, Jérémie Dubois-Lacoste, Thomas G Stützle, and Mauro Birattari. *The irace package: User guide*. IRIDIA, Institut de Recherches Interdisciplinaires et de Développements en . . . , 2016.
- [38] P Lafaye de Micheaux, Remy Drouilhet, and Benoit Liquet. The r software. *Fundamentals of programming and statistical analysis*, pages 978–1, 2013.
- [39] Leonardo Trujillo, Ernesto Álvarez González, Edgar Galván, Juan J Tapia, and Antonin Ponsich. On the analysis of hyper-parameter space for a genetic programming system with iterated f-race: L. trujillo et al. *Soft Computing*, 24(19):14757–14770, 2020.
- [40] Marius Stanescu, Nicolas A Barriga, Andy Hess, and Michael Buro. Evaluating real-time strategy game states using convolutional neural networks. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–7. IEEE, 2016.
- [41] Graham JG Upton. Fisher’s exact test. *Journal of the Royal Statistical Society: Series A (Statistics in Society)*, 155(3):395–402, 1992.

7. Appendix

7.1 IRace installation

1. Download R Software¹ from its official repository (the repository changes depending on the country in which we work)
2. Run the installer and follow the setup wizard.
3. Add R software to PATH by running:

```
C:\Program Files\R\R-4.1.3\bin\x64
```

For 32-bit version:

```
C:\Program Files\R\R-4.1.3\bin\i386
```

4. Install IRace from the R console by running:

```
install.packages("irace", repos = "https://cloud.r-project.org")
```

5. Run:

¹https://www.r-project.org.translate.google/?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=tc

```
C:\> Rscript -e "cat(gsub('/', '\\\\', system.file(package='irace', 'bin
```

It will output a path (replace x64 with i386 if you are on a 32-bit system) that you need to add to your PATH.

6. Verify the installation by running in your CMD:

```
C:\> irace --help
```

7. Create an Irace configuration folder in your project.
8. Create Parameter.txt with the parameters and their range.
9. Create Instance.txt with only one input: default.
10. Create target-runner.bat; this script serves as a bridge between Planet Wars and the Irace package.
11. Now we can execute our optimization by running:

```
irace --scenario scenario.txt
```