الجمهوريــة الجزائريــة الديمقراطيــة الشعبيــة

People's Democratic Republic of Algeria

وزارة الـتعلـيـم الـعالي و الـبـحــث الـعـلــمــي

Ministry of Higher Education and Scientific Research

Mustapha Stambouli University
Mascara

جامعـــة مصـطـفــى اسطمـبــولـــي
معسكــر

Faculty of Exact Sciences

كليــة العلــوم الـدقيـقــة

Computer Science Department

قســم الإعـــلام الألــــي

DOCTORAL THESIS

A thesis submitted in partial fulfilment of the requirements for the 3rd cycle (LMD) PhD degree in informatics

Option: Artificial Intelligence and Its Applications

Entitled:

Developing Intelligent Bots for Real-Time Strategy Games

***Presented by*: Abdessamed Ouessai**

On: 13/07/2022

Before the Committee:

President	Rochdi Bachir Bouiadjra	MCA	University of Mascara
Examiner	Mohammed Rebbah	MCA	University of Mascara
Examiner	Ahmed Louazani	MCA	University of Relizane
Supervisor	Mohammed Salem	MCA	University of Mascara
Co-Supervisor	Antonio M. Mora	Pr	University of Granada, Spain
Invitee	Antonio Fernández-Ares	MCA	University of Granada, Spain

Academic Year: 2021-2022

To my parents, Faffa Zerrouki and Bachir Ouessai

For the unconditional, unlimited support.

Acknowledgements

First and foremost, I would like to thank "Allah" for the strength bestowed upon me to keep moving forward.

This work would have never seen the light of day without the active support of my supervisor, **Dr Mohammed Salem**, whom I cannot thank enough for his passionate, charismatic, and human-first guidance. Thank you for tolerating my peculiar attitude.

My special thanks go to my co-supervisor **Dr Antonio M. Mora** who provided an uninterrupted stream of exceptional advice for four years, even during the most difficult times. **Dr Antonio** always kept his outstanding expertise within reach, never objecting to the length of my emails or my poor handling of deadlines. Thank you.

I express my utmost gratitude to the committee members for the time and efforts they spent reviewing this thesis, namely committee president **Dr Rochdi Bachir Bouiadjra**, **Dr Mohammed Rebbah**, **Dr Ahmed Louazani**, and **Dr Antonio Fernández-Ares**.

I would not miss the chance to thank my License supervisor **Dr Djelloul Mokadem** for inspiring my academic journey. I also extend my gratitude to **Dr Abderrahmane Bendahmane**, my Master's supervisor, for his role in structuring my research interests.

My thanks also go to the staff of the Faculty of Exact Sciences and the Department of Computer Science, at the University of Mascara, for the remarkable organizational and academic efforts.

Lastly, I would like to thank my parents and my siblings for their patience with me, without forgetting the three little distractors, *Bachirou*, *Sirine*, and *Nebras*.

Abstract

Real-Time Strategy (RTS) games impose multiple complex challenges to autonomous game-playing agents (a.k.a. bots), that also relate to real-world problems. The real-time aspect and the astronomical size of the decision and state spaces of an RTS game overwhelm the usual search algorithms. Monte-Carlo Tree Search (MCTS) was successfully applied in games featuring large decision and state spaces, such as Go, and was able to attain super-human performance in agents like AlphaGo and AlphaZero. Thus, researchers turned to MCTS as a potential candidate for solving RTS Games, and several RTS-specific enhancements were implemented, such as the support for real-time progression and combinatorial decisions. Nevertheless, MCTS is still far from replicating its Go success in RTS games. In this thesis, we propose several approaches to ease the RTS dimensionality burden on MCTS, in hopes of finding a path towards higher performance. To this end, we have made use of a detrimental-move pruning approach, proposed an integrated action/state abstraction process, and optimized its parameters through an Evolutionary Algorithm (EA). These approaches were tested and validated in the µRTS research platform, and the results showed moderate to significant performance gains. We expect the proposed approaches could be applied in commercial RTS games in the near future.

ملـخــص

إن ألعاب الاستراتيجية ذات الوقت الفعلي تفرض تحديات معقدة على برامج الذكاء الاصطناعي التي تحاول لعبها. هذه التحديات مرتبطة بشكل وثيق بمشاكل واقعية مثل صفة الوقت الفعلي، و ضخامة فضاءي القرار و الحال اللتان تعيقان خوارزميات البحث المعروفة. إن خوارزمية مونتي كارلو للبحث الشجري تمكنت من إيجاد حل لألعاب معقدة، مثل لعبة ڤو المتميزة بفضاء كبير للقرار و الحال. ذلك ما لفت انتباه الباحثين، و دفعهم لتكييف هذه الخوارزمية لأجل ألعاب الاستراتيجية ذات الوقت الفعلي. الشيء الذي أدى لظهور العديد من التحسينات الخاصة، كدعم الوقت الفعلي و فضاء القرار المركب. لكن بالرغم من ذلك لا تزال خوارزمية مونتي كارلو للبحث الشجري غير قادرة على تجسيد نجاحها في لعبة ڤو في ألعاب الاستراتيجية ذات الوقت الفعلي. من خلال هذه الأطروحة، نحاول تقديم بعض الطرق التي قد تفيد في تخفيف عبء ضخامة فضاءات البحث على خوارزمية مونتي كارلو للبحث الشجري، حيث قمنا باستعمال طريقة لتشذيب القرارات المضرة، ثم قدمنا اطارا تجريديا لفضاءي القرار و الحال، و قمنا بتحسين معايير هذا الاطار من خلال خوارزمية تطورية. النتائج المتحصل عليها بعد التجارب في منصة بحث خاصة بالألعاب الاستراتيجية ذات الوقت الفعلي، أظهرت تحسنا في الأداء بنسب تتفاوت بين المتوسطة و المعتبرة. نتوقع امكانية تطبيق هاته المقاربات في ألعاب الاستراتيجية ذات الوقت الفعلي التجارية في المستقبل القريب,

Résumé

Les jeux de stratégie en temps-réel (RTS) posent des défis considérables à l'encontre des agents autonomes (dites aussi, "bots"), des défis étroitement liés aux problèmes du monde réel. L'aspect temps-réel, et les énormes espaces d'état et de décision, accablent les algorithmes de recherche traditionnelle. L'algorithme Monte-Carlo Tree Search (MCTS) s'est réjoui d'un succès immense dans les domaines ayant de larges espaces d'état et de décision, tel que le jeu Go, où il a pu démontrer des capacités surhumaines à travers les agents AlphaGo et AlphaZero. Les chercheurs ont pris note, et ont visé à adapter MCTS pour les jeux RTS. Diverses améliorations spécifiques aux jeux RTS ont vu le jour, comme le support de l'aspect temps-réel et des décisions combinatoires. Néanmoins, MCTS reste incapable de reproduire son succès dans Go dans les jeux RTS. Dans cette thèse, on propose des approches qui tentent de diminuer l'impact de la haute dimensionalité sur MCTS, visant à atteindre des performances plus élevées dans le domaine des RTS. Pour cela, on a utilisé une méthode d'élagage des actions, et on a proposé un mécanisme d'abstraction d'états et d'actions intégré, qu'on a ensuite optimisé par un algorithme évolutionnaire. Ces approches ont été testées et validées dans la plateforme de recherche µRTS, et les résultats obtenus démontrent des gains de performance de degré modéré jusqu'à un degré considérable. On prévoit que les approches proposées pourraient être appliquées dans les jeux RTS commerciaux dans un proche avenir.

Contents

List of Figures

List of Tables

List of Abbreviations

1

Introduction

"A computer would deserve to be called intelligent if it could deceive a human into believing that it was human."

— Alan Turing

Ever since the dawn of humanity, games occupied a prominent position as a widespread sociocultural phenomenon. There is no culture or civilization that roamed the earth and did not engage in some form of ludic activity, regardless of the motives. People played games for different reasons, entertainment being the most notable one, but hardly the only one. Ruling elites and military leaders used board games as a safe way to sharpen and develop their strategic decision-making abilities. Non-elites were mostly interested in physically-challenging games that helped enhance their fitness and physical readiness for the serious tasks ahead. And of course, gambling games were a way for the lucky ones to make quick money, despite the questionable nature of such games. Clearly, games offered an enticing fusion of fun and benefit that not only kept them relevant, but also continued to amplify their prominence until this day.

Nowadays, with the broad adoption of digital platforms (PCs, Smartphones, Tablets, Smart TVs...etc.) digital games, also known as Video Games, emerged as this era's *de facto* form of games. Any digital platform can host digital games, and some platforms exist solely for this purpose (i.e. Game Consoles). Each year, a plethora of games are released and enjoyed by millions around the globe. The game industry is an ever-growing behemoth, counting thousands of game development studios, and thousands of supporting hardware, software, and artistic production companies. Huge budgets are allocated for the development of the next big franchises, and a wide array of cutting-edge technologies continuously contribute to their success. Games transitioned from a simple and quick, fun and profit activity, to a full-fledged medium of expression integrating art, science, and technology.

Other sectors are taking cues from the discipline of game design to make serious, dry, and often daunting tasks much more approachable, and even fun to do. Known as Gamification, this growing trend is becoming more ubiquitous in logistical industries, education, health, and military sectors, just to name a few. Moreover, anyone with a smartphone can easily find and interact with a host of gamified apps conceived to reduce the friction associated with doing unpleasant tasks, such as learning a new language, dropping detrimental habits, or engaging in physical activities.

1.1 | Games and Artificial Intelligence

Games helped in bringing forward and shaping Artificial Intelligence (AI) to reach its current state. Early computer science pioneers viewed cognitively-demanding board games, in the vein of Chess and Checkers, as the perfect environments to study machine intelligence. A machine was considered intelligent if it could play these games at a human-comparable level. One of the very first attempts to solve Chess was attempted by none other than Claude Shannon (Shannon, 1950), who devised a MiniMax approach using a manually designed approximate state evaluation function. Later, Turing (1953) proposed an approach to program computers to play Chess and Checkers. Both Alan Turing and Claude Shannon considered solving these games as an important first step towards tackling more significant machine intelligence problems. Decades later, Checkers has been solved (Schaeffer et al., 2007), Chess AI saw tremendous advances (Campbell et al., 2002), and the mindset of using games as a hatchery for AI techniques remained.

Academia gradually adopted this point of view, and research questions concerning more complex and more interesting games kept emerging, leading to the development of numerous AI techniques and frameworks, such as Reinforcement Learning (RL) (Sutton and Barto, 2018). With the democratization of digital games, AI researchers were attracted to the extent of possibilities offered by this new medium. Digital games offered a set of rich, complex, and controlled environments, perfect for use as experimentation grounds for a class of problems previously difficult to approach otherwise. Laird and VanLent (2001) issued a call to expand AI research through digital games, and expected the endeavor to eventually lead AI towards human-level performance. Since then, regular academic gatherings and publications took form around the subject of Games and AI, leading to an explosion in the number of research papers and academics studying the subject.

The game industry views AI differently. Employing AI within a digital game is usually driven by an economic need to increase the value of the game product, which would in-turn increase sales and profits. The goal of a professional AI programmer is not to advance the AI state-of-the-art, but to create artificial agents capable of generating good entertainment value that caters to players of all skill levels, while providing a high degree of immersion. In general, game industry professionals do not place too much trust in cutting-edge AI techniques developed in academia, citing their experimental nature that does not yet account for the strict technical and usability requirements of commercial game products. Still, if a technique is too useful to ignore, it will find its way into commercial games, albeit in a more adapted and less-experimental form.

For both sides, the utility of AI in games does not stop at developing highly intelligent/entertaining game-playing agents or Non-Playable Characters (NPCs). AI is also used to automatically generate game content (Levels, characters, art, items, ..., etc.) through Procedural Content Generation (PCG), and to understand and fine-tune game parameters by analyzing player behavior and emotions in play-testing phases, a practice known as Player-Modeling (Yannakakis and Togelius, 2018).

1.2 | Thesis Objective & Contributions

In line with the academic perspective of the relationship between AI and games, the objective of this thesis is to study and develop intelligent game-playing agents for one of the most difficult digital games subgenre for AI, namely Real-Time Strategy (RTS) games. As a research domain, RTS games succeeded in attracting a large pool of academic and corporate researchers, due to their many AI challenges closely related to real-world problems, and because of their complexity, many orders of magnitude above previous AI benchmark games (Chess and Go).

In this thesis, we will discuss some successful propositions developed for adapting and improving the emerging Monte-Carlo Tree Search (MCTS) holistic AI framework for RTS game-playing agents, through the use of domain knowledge. In particular, we will demonstrate that pruning detrimental moves from the search space could result in a sizable performance gain for MCTS-based agents (Ouessai et al., 2020a). We will also present a flexible action abstraction framework built on RTS heuristics and an action preselection process that precedes the execution of MCTS to frame and restrict the enormous decision space (Ouessai et al., 2020b). We will also show that our preselection framework can perform better if we optimize its parameters through an Evolutionary

Algorithm (EA) (Ouessai et al., 2022). All proposals were validated by experiments carried out using µRTS, a lightweight, fast, and research-focused RTS simulator.

The research work on this thesis resulted in several scientific contributions, including multiple communications and a journal publication, as shown in the following chronologically ordered list:

- **Communication:** A. Ouessai, M. Salem, and A. M. Mora, "Online Adversarial Planning in µRTS: A Survey" presented at the 2019 International Conference on Theoretical and Applicative Aspects of Computer Science (ICTAACS), Skikda, Algeria, Dec. 2019.
- **Communication:** A. Ouessai, M. Salem, and A. M. Mora, "Improving the Performance of MCTS-Based µRTS Agents Through Move Pruning" presented virtually at the 2020 IEEE Conference on Games (CoG), Osaka, Japan, Aug. 2020.
- **Communication:** A. Ouessai, M. Salem, and A. M. Mora, "Parametric Action Pre-Selection for MCTS in Real-Time Strategy Games" presented virtually at the VI Congress of the Spanish Society for Video Game Sciences (CoSECiVi), Madrid, Spain, Oct. 2020.
- **Publication:** A. Ouessai, M. Salem, and A. M. Mora, "Evolving action pre-selection parameters for MCTS in real-time strategy games" Entertainment Computing, vol. 42, p. 100493, May 2022.

1.3 | Thesis Outline

This thesis is structured around six chapters, including this introduction and a conclusion, plus an appendix. In the following, we provide a short description of the subjects treated by the subsequent chapters:

- **Chapter 2 - Background and State of the Art:** This chapter combines essential RTS background information with a listing of the most important works in the domain of RTS AI.
- **Chapter 3 - Move Pruning in MCTS:** In this chapter we describe a domain knowledge-based, detrimental move pruning approach applied to the MCTS framework.

- **Chapter 4 - Parametric Action Preselection:** Here, we introduce an action abstraction framework, based on RTS heuristics, that can effectively restrict and frame the RTS search space for subsequent planning or search algorithms, such as MCTS.
- **Chapter 5 - Evolutionary Action Preselection:** In this chapter, we employ an EA to optimize the parameters of our action preselection framework in various environments, against strong state-of-the-art agents.
- **Chapter 6 - Conclusion:** Summarizes all the approaches discussed in the thesis, and discusses other areas of applications, and future research perspectives.
- **Appendix A - ParaMCTS Parameters:** This appendix contains the full description of the parameters of our proposed action preselection algorithm, ParaMCTS.

2

Background and State of the Art

"Perception is strong and sight weak. In strategy, it is important to see distant things as if they were close and to take a distanced view of close things"
— Miyamoto Musashi

Digital games (henceforth, *games*) come in diverse forms and dimensions. A game is defined by its set of rules, challenges, and objectives, that distinguish it from other games. Games sharing a similar set of rules, challenges, and objectives are grouped together under the same category, known as the game **genre**. When new design variations arise within the same genre, creating significant dissimilarities between games of the same genre, subgenres are born and later grown into their own genres. Game aesthetics do not play a major role in genre definition. Genres evolved from classical games, and new genres emerge each time a new game could not fit the blueprints of the existing genres. According to Fencott (2012), 389 genre names were recorded in circulation, from which we can cite the most popular ones: Action, Adventure, Role-Playing, Sports, Simulators, Survival, Strategy, ..., etc. New genres, and hybrid genres, keep appearing; such as the PCG-heavy Roguelike, or the technical Soulslike.

RTS games initially emerged as a subgenre of Strategy games, distinguished by their real-time constraint responsible for several challenging ramifications, asymmetrically impacting human and artificial players. Nowadays, RTS games represent an independent genre spanning hundreds of games, several candidate subgenres, and professional e-sports leagues and tournaments. In this chapter, we define RTS games and their specific characteristics, and explain why they constitute an attractive AI research test-bed. We also provide a global view of the RTS AI state-of-the-art, with a focus on online adversarial planning techniques that succeeded in dealing with the complexities of RTS AI challenges (Ouessai et al., 2019).

2.1 | Real-Time Strategy (RTS) Games

RTS games follow a simple formula derived from the dynamics of veritable warfare situations. Essentially, an RTS game is a computer-based warfare simulator that offers the necessary means for players to embody the role of a filed-commander participating in an armed conflict of considerable scale. An RTS match involves at least two players representing opposing field-commanders, each trying to annihilate the opposing side's forces to achieve victory. To realize their objectives, players must initiate and develop an economy by exploiting resources previously discovered in the environment. The economy is then put to use to support training and enhancing an army, composed of heterogeneous units, which would engage in diverse military operations such as defending own bases or capturing enemy bases. A match takes place in an environment, known as the game map, representing the conflict zone. There can be several types of game maps, and each one may offer unique challenges and features that would force players to develop map-specific strategies. The events of an RTS match happen in real-time, which signifies that players do not take turns to execute their actions, instead, they may issue orders to their units at any point in time. Winning an RTS match requires a combination of judicious strategic planning and astute tactical maneuvering.

Figure 2.1 presents a screen capture of an RTS match in Microsoft's influential RTS title, AGE OF EMPIRES IV (2021). This is an instance of a medieval-setting RTS game, as can be seen from the weaponry and outfits of units. Figure 2.1's screen was captured in the midst of a confrontation between two army squads belonging to the two participating players. The figure also highlights a user interface control cluster, and a minimap. The latter is essential in keeping up with the events happening in every controlled zone of the game map.

The RTS formula tolerates a degree of variability to preserve originality among the different RTS games. For instance, the types and attributes of units and resources, the physical properties of the game map, and the overall degree of realism, are known to vary from one RTS game to another. The game setting holds an important and direct influence on game-specific variations. For example, a medieval setting (as shown in Figure 2.1) cannot include flying units, as in a modern or sci-fi setting. The game setting also dictates the number and characteristics of the available factions. A faction represents a distinctive set of units and structures, with unique attributes, abilities, and me-

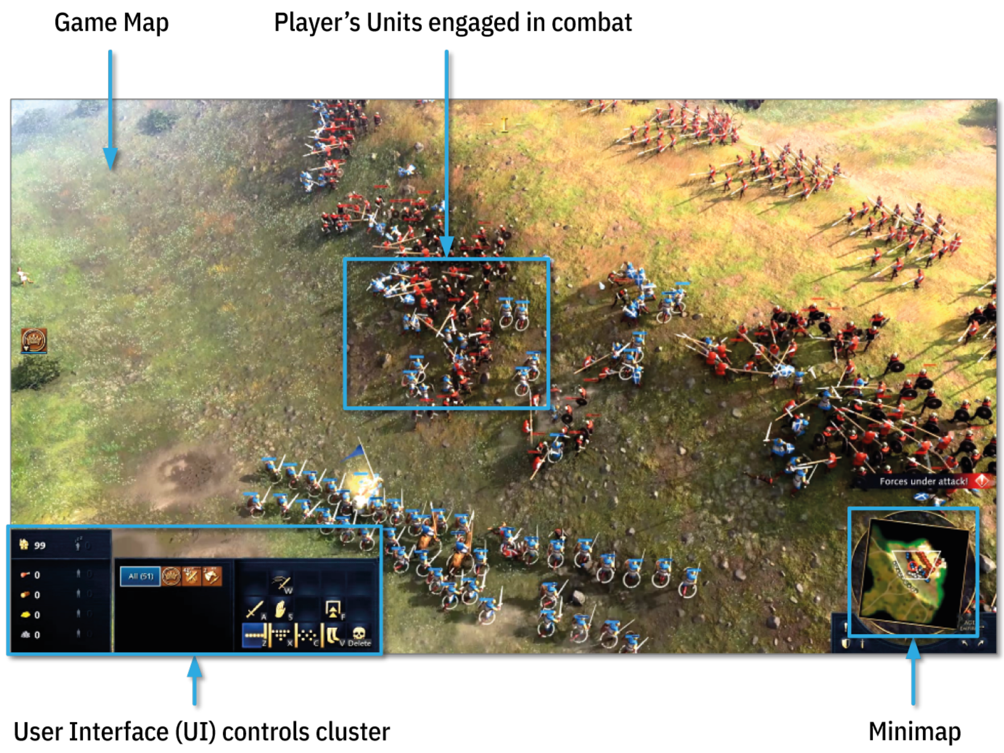

Game Map
Player's Units engaged in combat
User Interface (UI) controls cluster
Minimap
Forces under attack!

Figure 2.1: Screenshot of an actual PC RTS match from the popular RTS game AGE OF EMPIRES IV. This screenshot depicts a real-time battle unfolding between two players, identified by the color of their units (blue vs. red).

chanics. Opposing players may choose different factions in the same match (asymmetric factions), oppenning up new strategic and tactical pathways for every combination of opposing factions. For instance, AGE OF EMPIRES IV (2021) features eight factions, called civilizations, that include representations of The Abbasid Dynasty and The Holy Roman Empire. The RTS formula, and its space of variations, came as the result of years of experiments and refinements, as we will see next.

2.1.1 | Origins & Evolution

Original strategy board games, in the vein of Chess, Checkers, Shogi, and Go, are turn-based by nature. In such games, players can afford the luxury of stress-free careful planning. Playing these games in real-time, within the constrained physical medium of boards and pieces, does not make sense. Early digital strategy games, such as INVASION (1972) and COMPUTER BISMARK (1980), adopted the turn-based system from board games (Walker, 2002), but it was clear that the new medium could support more interesting play styles. The first strategy games to experiment with a real-time component were UTOPIA (1981) by Don Daglow, and CYTRON MASTERS (1982) by Danielle Bunten. The first sought to blend real-time events with turn-based cycles, and the second limited real-time interactions to short-term tactical planning sequences. These games symbolized a proof of concept that led the way to more interesting developments (Barton, 2007; Moss, 2017).

Dan Adams (2006) affirm that THE ANCIENT ART OF WAR (1984) by Dave and Barry Murry was the very first fully real-time strategy game, followed later by Technosoft's HERZOG ZWEI (1989) (Sharkey, 2004). Both games helped in forming the first RTS games template, but the influence of HERZOG ZWEI was more significant due to its introduction of many of the genre's distinctive mechanics such as unit training, resource management, and base building and destruction. HERZOG ZWEI is considered as the precursor of modern RTS games, and is often credited for inspiring the emergence of the acclaimed RTS game, DUNE II (1992) (Clarke-Willson, 1998; Sharkey, 2004). Westwood's DUNE II was the culmination of the preceding technical and creative efforts, it successfully combined the fledgling RTS mechanics scattered across prior pre-RTS games, and presented them in a remarkably coherent and modern package. DUNE II was the first game to actually use the "Real-Time Strategy" term, coined by its producer Brett Sperry, to define its genre (Geryk, 2011). This last development ushered in the mainstream popularization of the RTS subgenre as a standalone genre.

Following the success of DUNE II, and the solidification of the fundamental RTS formula, several inaugural titles of today's big RTS franchises materialized in the period of the nineties (Dan Adams, 2006; Walker, 2002). Blizzard Entertainment released the fantasy RTS WARCRAFT: ORCS & HUMANS (1994), which served as the basis for their second RTS, STARCRAFT (1998), featuring a Sci-Fi setting and vastly improved gameplay mechanics. Westwood's next RTS attempt developed into the highly influential COMMAND & CONQUER (1995). Ensemble Studios sought to create a slow-paced RTS game by introducing the concept of technological ages in AGE OF EMPIRES (1997). The transition from two-dimensional (2D) to three-dimensional (3D) graphics engines opened up new perspectives for the RTS genre. It became possible to integrate realistic physics, larger game maps, higher maximum units count, and more features. HOMEWORLD (1999), one of the first 3D RTS games, took the genre to an interesting, fully 3D, outer-space warfare setting with six degrees of freedom.

Nowadays, with the widespread availability of powerful microchips and sophisticated game engines, RTS game developers do not shy away from harnessing the full potential of advanced technologies to dramatically raise the scale and fidelity of their games. One striking example is the immense-scale and complexity of Creative Assembly's TOTAL WAR (2000-2022) RTS franchise. A noteworthy RTS subgenre known as Multiplayer Online Battle Arena (MOBA) focuses on the tactical control of a single unit in a fast-paced confrontation between other players' units, across the network. This subgenre was born as a custom mode for WARCRAFT III (2003), called DEFENSE OF THE ANCIENTS (DoTA), then matured as a proper genre with games like LEAGUE OF LEGENDS and HEROES OF THE STORM (Minotti, 2014).

It is worth mentioning that RTS games remain predominantly Personal Computer (PC) games. The mouse and keyboard combination was critical in facilitating the complex control scheme demanded by RTS games. Today's most important RTS games includes the following: STARCRAFT II, TOTAL WAR WARHAMMER series, AGE OF EMPIRES IV, COMPANY OF HEROES 2, and ASHES OF SINGULARITY.

2.1.2 | RTS Elements

RTS games represent a system of several interacting components and ideas that work in harmony to create the experience and pressure of a warfare situation. Under this section, we briefly describe the core components and ideas that constitute an RTS game (Adams, 2014b). Figure 2.2 presents a screenshot of STARCRAFT II, highlighting some RTS elements discussed below.

2.1.2.1 | Units & Structures

Sometimes collectively referred to just as units, units and structures are the smallest controllable entities in an RTS game. One may compare units to Chess pieces, although with a more sophisticated control scheme and some degree of autonomy. A unit is typically mobile and is produced by some structure. A structure is a fixed building that is built by specialized units, and offers unit training and upgrading facilities, among many possible strategic advantages depending on its type. Examples of units and structures are shown in Figure 2.1 and Figure 2.2. Units and structures may be atomic, or compound, and are distinguished by a list of attributes that define their behavior and type. Common unit attributes include: *Health*, *Weapon*, *Damage*, *Accuracy*, *Offensive Range*, *Vision Range*, *Fire Rate*, *Speed*, *Size*, *Defensive Dodging*, ..., etc (Adams, 2014b). Upon confrontation, opposing units deal damage to each other by unit-specific means. A unit or structure will be removed from the game if it has sustained enough damage equal or greater than its Health Points (HP).

RTS players usually deal with units of different special abilities, that can affect the tactics and strategy of play if correctly employed. As examples of units' special abilities, we cite the following:

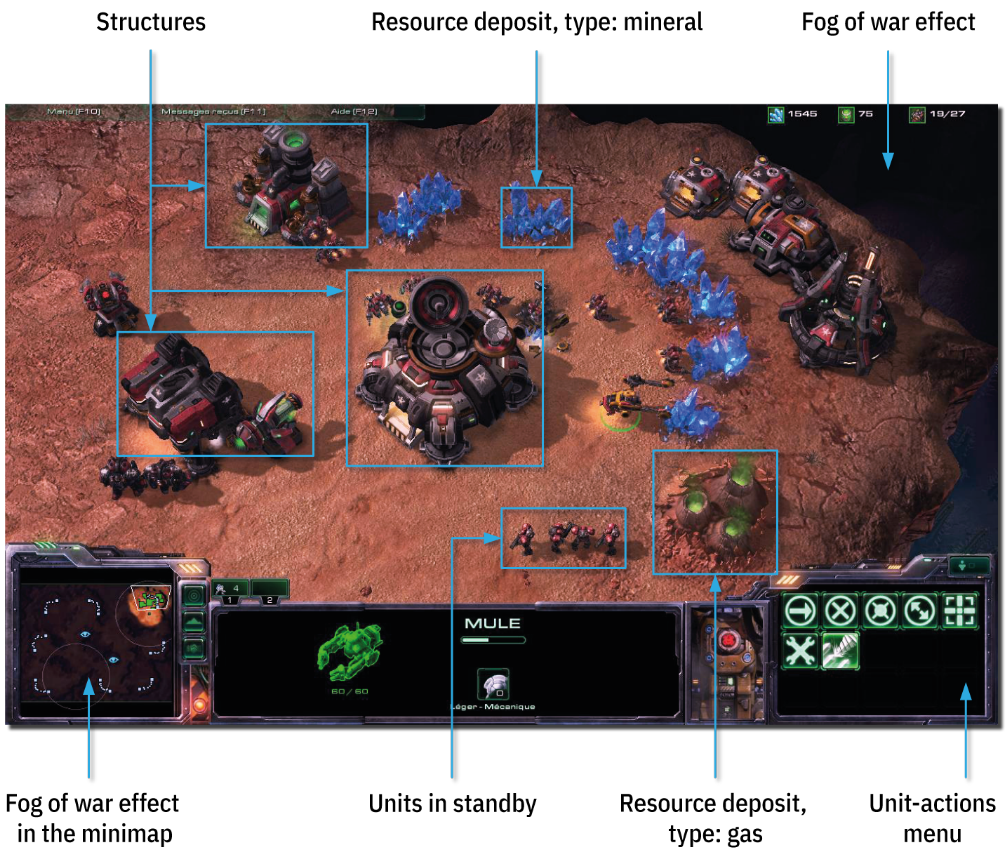

Structures
Resource deposit, type: mineral
Fog of war effect
Fog of war effect in the minimap
Units in standby
Resource deposit, type: gas
Unit-actions menu
MULE

Figure 2.2: A STARCRAFT II screenshot showing the early state of a player's base in an RTS match. The player is focused on gathering resources and training assault units, as seen by the concentration of worker units around resource deposits, and the units in standby.

- **Stealth:** The unit may become invisible to enemy units, facilitating sneak tactics.
- **Flying/Sailing:** Grants the possibility to travel through terrain inaccessible to other units.
- **Repair/Heal:** The ability to restore the lost HP of allied units.
- **Transport:** Makes it possible to rapidly convey a group of units across the map.
- **Constructing/Training:** The capacity to build structures or train new units.
- **Leadership:** The unit can enhance key combat attributes of nearby units.

2.1.2.2 | Indirect Control

RTS players control their units in an indirect fashion, by issuing orders in the form of unit-actions to each. This control scheme is important because it allows players to maintain granular control over a high number of units, whereas direct control is best-suited for controlling a single unit at a time. Upon receiving an order, a unit acts autonomously to execute it, regardless of the type of the unit-action it represents. How the unit behaves to execute a unit-action is dictated by its attributes, and the internal implementation details of the game. The following is a list of the most common unit-action orders possible in an RTS game (Adams, 2014b):

- **Move:** Orders the unit to move to a chosen location on the map. The unit usually uses a pathfinding algorithm to find its way to the destination. The unit may attack enemy units found in its way.
- **Attack:** Initiate an attack on a designated enemy unit. If the target is out of range, the unit advances towards it.
- **Stop:** Halts the execution of any action.
- **Hold:** The unit maintains its current position, attacking any unit that may enter its offensive range.
- **Retreat:** Disengage from any combat situation and return to safety.
- **Dash:** A quicker Move variant that ignores any enemy units on the way.
- **Patrol:** Move back and forth between two waypoints and attack any approaching enemy unit.
- **Produce:** Specific to factory structures. Produce units in exchange for an amount of resources.
- **Adopt formation:** Specific to a group of units. Assume a special tactical formation in hopes of gaining an advantage.

In Figure 2.2, one may note a unit-actions UI panel containing buttons specific to each unit-action. Modern RTS games include squad management capabilities that simplify the task of controlling groups of units, by allowing a group to execute a shared order, instead of having the player manually issue the same order to each group member.

2.1.2.3 | Resource Economy

Resources act as a currency within an RTS game, and form the basis of the player's economy. Usually, players order special worker units to keep gathering resources from natural resource deposits until depletion. To train new units and construct structures, a player must spend a set amount of resources in exchange, relative to the value of the new unit/structure. There can be multiple types of resources and a future unit/structure may require different quantities of different resource types. The game setting defines the types of resources available. For instance, medieval settings may use gold, stone, and wood, as in the AGE OF EMPIRES series. A sci-fi setting as in the STARCRAFT series, use minerals and "vespene" gas as shown in Figure 2.2, to depict resources. Some RTS games do not use physical resources, but instead, resort to a resource points accumulation mechanism which grants points by holding strategic locations in the map, as in COMPANY OF HEROES.

Controlling resource-rich points and interrupting the resource supply lines of opposing players are two examples of using resources to gain a strategic advantage.

2.1.2.4 | Technology Trees

Over the course of an RTS match, players may have the opportunity to gradually upgrade the attributes of their units, or unlock additional unit types, structures, or abilities, in exchange for resources, or by some other game-specific mechanism. These upgrades and unlocks are usually structured as a dependency tree, where one upgrade/unlock leads to new possible upgrades/unlocks. Such a tree is known as the *Technology Tree*, and may host many upgrade paths for a player to take from root to leaves, depending on his strategic approach. An example of a technology tree of moderate complexity, concerning the Protoss faction of STARCRAFT II, is shown in Figure 2.3. The leaves of the tree are usually reserved to some high-impact, game-altering abilities or units, like an atomic weapon. Technology trees bring an additional layer of strategic complexity to RTS games, although they do not constitute an RTS-specific element. Technology trees are also common in Role-Playing Games (RPGs).

2.1.2.5 | Game Map

The physical space where an RTS match unfolds is known as the game map. The game map is a major factor in determining the strategic pathways players can take. It is an arbitrarily-sized geographical zone that may include geological or urban features establishing strategic locations such as choke points, hideouts, and elevated planes. In

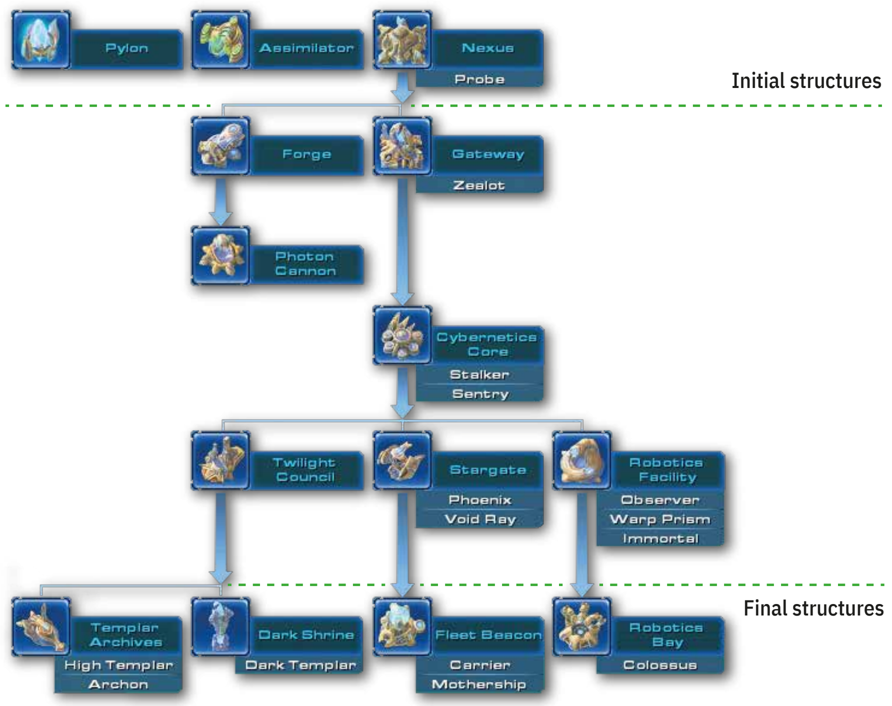


Figure 2.3: An annotated technology tree regarding STARCRAFT II's Protoss faction. Arrows depict a dependency relationship between two structures. A Protoss player starts with three possible structures, and progressively unlocks the structures bellow. Unlocked units are shown beneath structures. Note that technology trees are usually faction-specific.

addition, the game map may host natural resource deposits scattered across specific locations, and may also include neutral forces that can be leveraged in some strategic manner. Furthermore, a game map can be dynamic, meaning that abrupt layout changes may happen during an RTS match, which would force players to shift their strategies. The design of a game map is the responsibility of level designers within a game-development team. For further clarification, Figure 2.4 shows four different RTS game maps, featuring distinct layouts and features.

It is possible to draw a sharp contrast between RTS games and board games in this regard. Board games usually possess a single, static board (equivalent to a game map), with well-known strategies. Whereas in RTS games, the numbers and configurations of possible maps is unlimited, with each map requiring a different strategic approach.

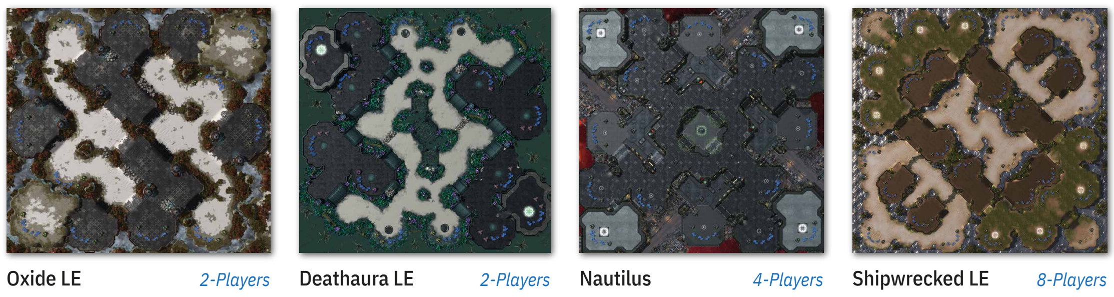


Figure 2.4: A selection of official and user-made STARCRAFT II competition maps. Each map has an attached name for identification, and each supports up to the indicated number of players. Note the symmetry of layout in each map, which is a way to establish a correct game balance.

2.1.2.6 | Fog of War

In a veritable warfare situation, there is always an element of uncertainty and doubt at play, whether about the location and strength of enemy forces, terrain conditions, or any unforeseen events. Military strategists call this warfare uncertainty the fog of war. RTS game designers model the fog of war as a dark map overlay concealing all parts of the game map that the player has yet to explore. A concealed zone is fully revealed once a player unit traverses through it, but may revert to only conceal enemy movements on the exit of the player's unit. A unit's range of vision attribute determines how far it may see through the fog of war. Some structures (e.g. Radar or Sentinel towers) may be constructed to keep a limited zone fully visible. In Figure 2.2, two examples of the effect of fog of war are shown.

2.1.2.7 | User Interface & Camera

With the evolution of RTS games and the growth of their complexity, traditional control methods using game-pads and joysticks would not keep up. The versatility of the PC's mouse and keyboard input combination attracted RTS designers to develop their games on the PC platform. Because of this early choice, the PC remains as the platform of choice for RTS games. These games gained a PC-friendly user interface and control scheme reminiscent of a typical PC software. Players would control their units by selecting them and clicking buttons and toggles laid out in a Heads-Up Display (HUD)-like interface, as show in Figure 2.1 and Figure 2.2. Support for keyboard shortcuts enabled advanced players to speed up their game and develop rapid micromanagement skills. This model of interaction is known as the multipresence model (Adams, 2014b), because a player can control multiple aspect of the game at once.

Player-assisting UI features include a minimap that shows a condensed overview of the game map with units' locations, accounting for the fog of war status. It is also common to find resource and unit counters, event notifications, detailed views of unit and economy statistics, and much more, depending on the RTS implementation.

In most cases, an RTS player has a top-down aerial view of the game map, from where he could obtain a clear overall perspective of the situation. It may be possible to adjust the angle, position, and tilt of the game's camera to focus on specific zones.

2.1.3 | Challenges *for Humans*

Every game is expected to offer at least one type of challenge to its potential players. A challenge is a gameplay element that represents a task to be completed to advance through the game. Each type of challenge targets a different player skill. Challenges are usually organized hierarchically, with the top-most challenge alluding to the game's victory condition. To attain the victory condition, a series of intermediate challenges must be completed, which are themselves composed of atomic challenges (Adams, 2014a). Challenges are not required to be explicit. Oftentimes, it is up to the player to derive intermediate challenges from the victory condition. This is the case for RTS games, where intermediate challenges change depending on the game map and the opponent. Nevertheless, the atomic challenges of RTS games remain largely the same. The list that follows enumerates the most important types of atomic challenges found in RTS games, as adapted from (Adams, 2014a,b):

- **Strategic conflict:** Being in a war simulation, a player is expected to exhibit similar skills to those of a filed commander. The challenge of strategic conflict requires careful situational analysis followed by the elaboration of a functioning plan of action. Challenges nested within any conflict situation include the survival of units and the progressive reduction of enemy units. The challenge of conflict logistics is also important, and pertains to the maintenance of a steady supply of combat units, while protecting production facilities and resource supply routes.

- **Economic:** The limited supply of resources, and the need to continuously produce units and build the tech tree, drive players to carefully consider how, where, and when to invest resources to maximize the chances of winning.

- **Exploration:** Acquiring information early about the location of resource nodes, strategic points, and enemy outposts, could grant a significant strategic advantage. The player must seek out such information through systematic exploration of the game map while minimizing the risk of being detected.

2.2 | RTS Games & AI

AI technology is employed as part of an RTS game in two ways. The first, as a game-enhancing element included to serve players in some form. The second way, that is most relevant to this thesis, is by using experimental AI techniques in an attempt to solve RTS games, and advance the AI state-of-the-art (Yannakakis and Togelius, 2018). In the latter case, RTS games may also serve as an AI test-bed.

2.2.1 | RTS AI in the Industry

The game development industry views AI as a technology that helps in adding value to their final products. A game of high value provides high-quality entertainment, and this is an area where AI does a good job by providing believable artificial opponents and NPCs. The main AI requirement of RTS game development companies is to provide artificial opponents with scalable difficulty levels, such as to remain enjoyable to all RTS players, regardless of their skill level. In that regard, AI programmers must create agents having a degree of predictability in order to maintain coherence with all other game components, whilst embedding a human-like play aspect to increase the AI's believability.

The AI techniques used in the game industry usually combine rule-based agents with state-transition models such as Finite-State Machines (FSMs) or Hierarchical Finite-State Machines (hFSMs), or decision-making formalisms like Decision Trees or Behavior Trees (Kirby, 2011; Millington, 2019). Goal-oriented planning systems and rule production systems are also prevalent (Schwab, 2009). One common approach is to model interactions between different units and components, in hopes of creating seemingly intelligent emergent behaviors (Rabin, 2007). The first usage of computational intelligence techniques like Neural Networks in a commercial RTS game appeared in BLITZKRIEG 3 (O'Connor, 2017). Commercial AI agents are often allowed to cheat, by having access to more information than the human player or allowing impossible maneuvers, as a way to scale-up difficulty (Rabin, 2007; Sun et al., 2018).

2.2.2 | RTS AI Research

The goal of RTS AI researchers is to develop intelligent agents capable of playing RTS games at a level similar to that of professional human RTS players, or beyond. The motive is to advance AI's state-of-the-art, and move closer toward understanding how to achieve human-level performance in similar tasks. Michael Buro was first to recognize the relevance of RTS AI, and issue a call (Buro, 2003b, 2004) to exploit the RTS domain as an AI research platform. Since then, the genre caught the attention of AI researchers, and interesting contributions began to appear (Ontañón et al., 2013). Competitions organized between RTS AI researchers aided in keeping a high interest, and in defining future research directions. STARCRAFT emerged as an RTS AI competition platform (Churchill et al., 2016). Later, specialized RTS AI research platforms, like μRTS[1], developed their own competitions (Ontañón et al., 2018).

From an AI standpoint, developing an intelligent game-playing agent for an RTS game is a very challenging task to undertake (Buro, 2004; Ontañón et al., 2013). This difficulty rises as the result of a combination of distinct factors, related to the unique mixture of characteristics in RTS games. We enumerate these factors in the following list:

- **The Real-Time Aspect:** RTS games run typically at 10 to 60 frames per second, creating a very short decision cycle for an AI to work with. If we suppose a game runs at 24 frames/s, this means a state change occurs every $1/24 = 42ms$, corresponding to the time left for an agent to compute a decision.
- **Simultaneous Moves:** Opposing players can issue commands to their units at the same time, in contrast to turn-based games.
- **Durative Actions:** A unit can take more than one decision cycle to complete the execution of certain commands. Thus, the effect of an action is not immediate.
- **Partial Observability:** In most RTS games, the game map is partially visible to the player due to the effects of the fog of war. In such circumstances, an agent is left with incomplete information about the game state.
- **Non-Determinism:** The effect of executing the same action can vary between two executions because of some possible stochastic parameters.

2.2.3 | Challenges *for AI*

Facing such a complex problem domain, a divide and conquer approach towards building an RTS game-playing agent should be adopted. By breaking down the problem into manageable sub-problems and solving each one separately, a bot/agent can be built through integrating the solutions found for each sub-problem. The main challenges facing artificial RTS game-playing agents can be categorized as follows (Ontañón et al., 2013; Robertson and Watson, 2014):

[1]`https://github.com/santiontanon/microrts`

- **Adversarial Planning:** How to determine the optimal sequence of actions leading to victory, when facing an opponent trying to do the same thing, in real-time. Online adversarial planning is an active research domain where techniques such as game-tree search and machine learning are employed. The methods proposed in later chapters focus on overcoming this challenge.
- **Learning:** How can an agent employ machine learning techniques to learn to play the game. The agent may use prior replay data, data acquired in-game, or from another game.
- **Uncertainty:** Imperfect information about the game state can negatively affect the agent's behavior. The agent has to minimize uncertainty in order to plan effectively. Uncertainty originates from the lack of sufficient information about the environment and/or the opponent's behavior.
- **Spatial Reasoning:** Where to place structures, units, and expand bases, strongly impacts the agent's strategy outcome. Spatial information is usually inferred from a terrain analysis module, and used to optimize placements and tactical group reasoning.
- **Temporal Reasoning:** When to execute an action is just as crucial as the "where", especially in the presence of durative actions. The effect of some actions will not manifest until later in the game.
- **Domain Knowledge Exploitation:** How to extract and use data from the available wealth of knowledge, in order to improve the agent's behavior. Knowledge is represented by expert replay logs, strategy guides, and expert strategies encoded in scripts.

2.2.4 | Levels of Abstraction

RTS AI researchers classify techniques according to the level of abstraction they operate on. A technique may need to deal with more than one challenge, which makes it hard to use challenges as categories. Three levels of abstractions are recognized by the RTS AI community: Strategy, Tactics, and Reactive Control (Ontañón et al., 2013, 2015; Robertson and Watson, 2014). Figure 2.5 was adapted from (Ontañón et al., 2015), and shows the sub-problems within each abstraction level. Each sub-problem is related to at least one challenge, and one sub-problem may depend on another sub-problem, either in the same level or in a different level. The behavior switch duration estimate shown

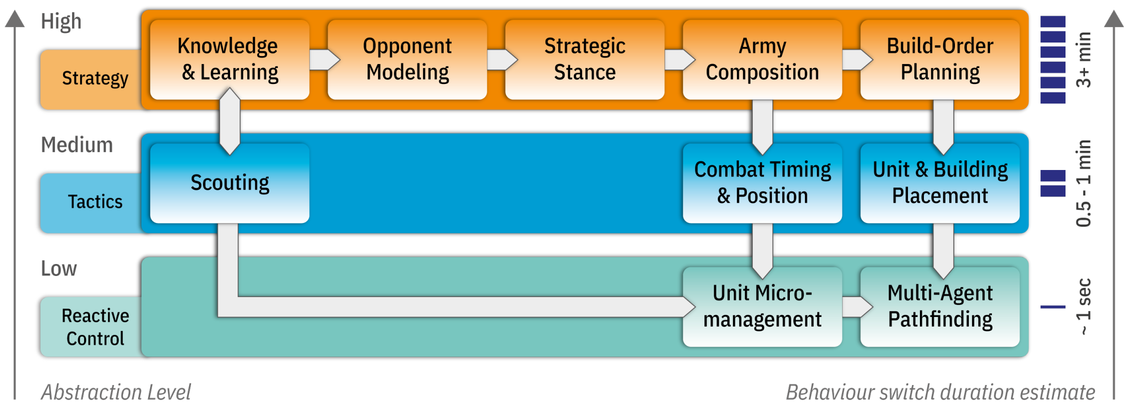


Figure 2.5: RTS sub-problems distributed according to the level of abstraction they operate on. Arrows represent dependency.

in the right axis is an approximation of the duration needed for the effects of an action to manifest in each level. Switching strategic stance, for example, would require at least three minutes for the result to take effect. The effect of unit micromanagement behavior switch, by contrast, manifests almost immediately. These estimates concern STARCRAFT.

Human players refer to the strategy level, and some parts of the tactics level, as macromanagement. Reactive control and the remaining parts of the tactics level are known as micromanagement (Ontañón et al., 2013). Next, we will give a brief description concerning each level.

2.2.4.1 | Strategy

In this level, the agent must consider the overall situation of the game and devise a plan of action to counter the perceived opponent strategy. The strategic decisions taken here have a long-term effect on the game. For example, in the build-order planning sub-problem, the agent has to find an optimal order for building the required army composition in the shortest time possible (Churchill and Buro, 2011; Churchill et al., 2019). From the research results regarding this sub-problem we can cite the work by Köstler and Gmeiner (2013) that used a multi-objective Genetic Algorithm (GA) to optimize build orders, and Tang et al. (2018)'s RL approach for learning optimal build orders.

2.2.4.2 | Tactics

Tactics refer to the intermediate decisions concerning groups of units tasked with the execution of the decisions taken at the strategic level. Tactical decisions affect the game in the medium-term, and usually relate to the questions of where and when to execute an action. Research in this direction may focus on building placements, as done by Barriga et al. (2014), when they used a GA to find optimal building locations. Tactical decisions may require the use of terrain analysis (Perkins, 2010; Richoux, 2022) techniques to assist in finding interesting map locations.

2.2.4.3 | Reactive Control

In the lowest level of abstraction, the goal of the agent is to efficiently execute tactical orders through precise unit maneuvering (micromanagement). Decisions here affect the game in the short-term, at the unit-level, and usually concern the behavior of units within a direct conflict situation. Lookahead-based search techniques are widely used for this task, as done by Churchill and Buro (2013). Evolving neural controllers for micromanagement is also an interesting application (Gabriel et al., 2012; Zhen and Watson, 2013).

2.2.5 | Research Platforms & µRTS

Developing intelligent game-playing agents within commercial RTS games can be a daunting endeavor, due to the systematic absence of official Application Programming Interfaces (APIs), and because of the limits imposed by the very nature of a product not built to support AI research. This situation led to the development of independent solutions such as the unofficial StarCraft API known as Brood War Application Programming Interface (BWAPI), which opened the door widely for AI research on StarCraft. Similarly, platforms entirely dedicated to simplify RTS AI research have emerged, such as ORTS (Buro, 2003a), Wargus[2], µRTS (microRTS) (Ontañón, 2013), ELF (Tian et al., 2017), and DeepRTS (Andersen et al., 2018). It is worth mentioning that in 2017, Blizzard Entertainment and DeepMind released a set of tools and an official API[3] for AI research in StarCraft II (Vinyals et al., 2017).

Each platform is best suited for attempting to solve a specific challenging, or experimenting with a certain technique. In this thesis we used µRTS as our experimentation platform because of its affinity with respect to online adversarial planning research.

[2]https://github.com/Wargus/wargus
[3]https://github.com/deepmind/pysc2

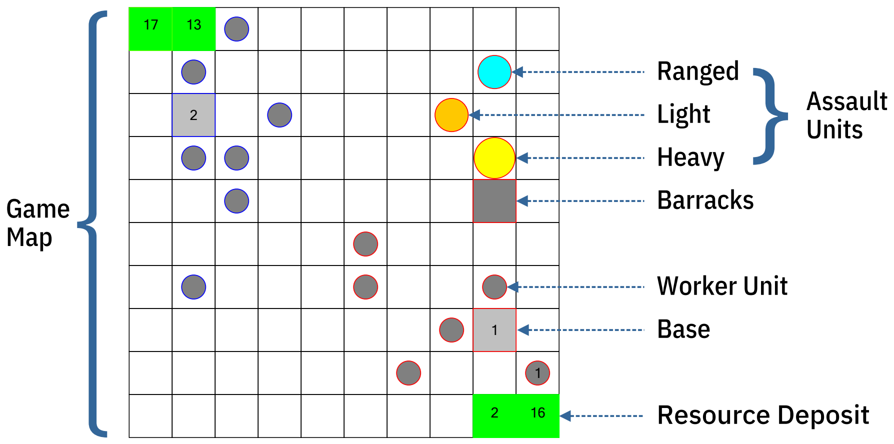


Figure 2.6: A graphical visualization of a µRTS match in a 10 × 10 map. Players can be distinguished by the outline color of their units (red and blue). Numbers indicate the amount of resources held in a base or by a worker, or the remaining resources in a deposit.

2.2.5.1 | µRTS

µRTS is an RTS AI research platform conceived by Santiago Ontañón (Ontañón, 2013) to address the necessity of a specialized open platform that captures all the basic aspects and proprieties of an RTS game. It provides the necessary facilities to quickly develop and test intelligent agents. The game component of µRTS is an abstract, minimalistic version of an RTS. All the basic rules and mechanics of a typical RTS game apply, and being open-source, it is freely customizable and extensible. µRTS is currently one of the most active RTS AI research platforms, mainly because of its lightweight, yet complete, feature set and its efficient built-in forward model. These distinctive characteristics allowed for the inception of many interesting holistic and hybrid planning approaches that deserve a special attention.

In the default µRTS game settings, shown in Figure 2.6, each player controls two types of structures: *base* and *barracks*, and four types of units: *worker*, *light*, *heavy* and *ranged*. The base produces workers and the barracks produces assault units, in exchange for a resource cost. All units can attack opponent units and structures, and workers can harvest resources. There is a single type of resources available. The game takes place on a map that consists of a 2D grid of arbitrary size, and each unit, structure, or resource deposit, occupies a single square. µRTS comes with a set of maps in different sizes and a map editor. The game can be configured to be partially or fully observable, deterministic or non-deterministic.

The lightweight characteristic of µRTS enables fast computation of state transition, which constitutes the basis for its forward model. The availability of a forward model permits the implementation of game-tree search methods that need to execute an important number of simulations. In fact, a good chunk of the research done on µRTS revolves around game-tree search algorithms. An area less-explored in other RTS games due to the absence of an efficient forward model.

2.2.6 | Formal Definition & Complexity

A typical RTS game can be seen as a two-player, non-deterministic, zero-sum (either win, or lose) game with imperfect information. The complexity of an RTS game can be estimated by determining its state space dimensionality, and branching factor. The number of possible game states within an RTS is enormous, if we compare it to traditional AI benchmark games such as Chess or Go. The state space of a regular StarCraft setting has been estimated around 10^{1685} whereas in Chess, 10^{47} and in Go, 10^{171}. Similar proportions are observed in the branching factor, where the average number of actions that can be executed at a decision point in StarCraft reaches 10^{50}, while it is estimated to be 36 in Chess and 180 in Go (Ontañón et al., 2013).

The reason for which RTS games possess such an astronomical state space and branching factor resides in the combinatorial structure of both state and action spaces, where a player can issue commands to multiple units at the same time, in an arbitrarily sized game map. Because non-determinism and imperfect information fall outside the scope of this thesis, we use the definition of a deterministic, perfect-information RTS game, as put forward by Ontañón (2017). Thus, an RTS game is defined as a tuple:

$$G = (S, A, P, \tau, L, W, s_{init}) \tag{2.1}$$

Where:

- S : The set of all possible states (state space).
- A : The set of all player-actions (decision space).
- P : The set of players. $P = \{max, min\}$ for two players.
- $\tau : S \times A \times A \rightarrow S$: The state transition function. It takes a game state at time t and a player-action for each player, then returns a new game state at time $t + 1$.
- $L : S \times A \times P \rightarrow \{true, false\}$: Determines the legality of a player-action in a given state for a specific player.

- $W : S \to P \cup \{ongoing, draw\}$: Determines the winner, if any, or whether the game is a draw or still ongoing.
- $s_{init} \in S$: The initial state.

Note that a player-action is a combination of unit-actions.

2.3 | Online Adversarial Planing: State of the Art

The goal of an RTS game-playing agent, or bot, is to generate relevant actions based on the current game state, under a limited computation budget. The agent is expected to generate a player-action for every game cycle, and the series of generated actions should lead the agent to victory against an arbitrary opponent. In other words, the agent has to figure out a winning plan in real-time, under the stress of a large state and action spaces, and a tight computation budget. This problem description aligns with the online adversarial planning problem, an active research area wherein µRTS is the most suitable platform. The contributions presented in later chapters are attempts to address the online adversarial planning problem in µRTS.

Online adversarial planing approaches can be placed under three categories (Ouessai et al., 2019): Low-Level Planning (LLP), High-Level Planning (HLP), and Hybrid Planning (HyP).

2.3.1 | Low-Level Planning (LLP)

Planning in the original state and action spaces is considered a Low-Level Planning (LLP) process, because of the low degree of abstraction considered. LLP directly confronts the complexity of the unmodified search space, and risks generating suboptimal plans when the space's dimensionality is too high. LLP is generally associated with game-tree search algorithms, and is known to produce good short-term tactical performance.

A game-tree, as shown in Figure 2.7, is a game representation using a tree structure, where nodes and edges represent states and actions, respectively. The root of the tree represents the initial game state and the execution of an action by a player will result in a new node with a new state. Each level (ply) in the tree is associated with one of the players, in an alternating fashion. Leaves of the tree represent terminal states, where a winner can be determined. In theory, if we can generate the complete game-tree, optimal actions can be derived. However, in practice, this is not feasible, due to the ex-

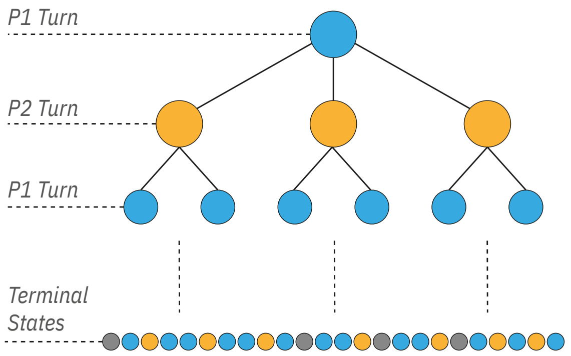


Figure 2.7: A game-tree representation for a two-player turn-based scenario. Terminal states indicate a win by either players, or a draw in case of gray states.

ponential size of the game-tree in RTS games, making traditional tree-search algorithms less effective. Even so, searching a partial game-tree is more plausible.

Because of the possibility to work on small scale RTS scenarios in µRTS, with a relatively low branching factor, game-tree search algorithms are applicable. Since such algorithms are rooted in the domain of turn-based games, they need to be adapted for real-time scenarios. This is what Ontañón (2012) attempted. By using the MiniMax game-tree search algorithm as a basis, he first defined the game-tree for real-time scenarios, where moves can occur simultaneously. Instead of alternating player nodes, the new tree structure permits successive nodes from the same player and different player nodes in the same level. Durative actions are dealt with similarly as in the Alpha-Beta Considering Duration (ABCD) algorithm (Churchill et al., 2012), which consists of forwarding states to the moment an agent is able to take action. This algorithm is known as Real-Time MiniMax (RTMM).

Search-space sampling based approaches are well suited for domains with a huge decision space dimensionality, operating under tight computation constraints. MCTS (Browne et al., 2012) is a family of sampling-based, anytime search algorithms, relying on Monte Carlo simulations to improve the reward estimate of actions. Each node holds a reward estimate and a visit count. The standard MCTS algorithm follows a four-step process: (1) select the node to explore from the current state, following a tree policy, (2) expand the selected node, (3) simulate the game (perform a playout) starting from the expanded node, following a default policy, and (4) back-propagate the simulation's outcome (reward) to the parent nodes by updating their reward estimates and visit counts.

Different MCTS algorithms differ mostly in the tree policy employed (Browne et al., 2012). In the simulation phase, the players' actions are chosen according to a default policy, until reaching a terminal state where the game outcome is clear (win, loss or draw). Or in most cases, until a playout time budget is consumed, after which, an evaluation function is applied to the intermediate state to determine its value. Reward estimates are improved iteratively, and usually the action leading to the most visited node is returned.

Upper Confidence bounds for Trees (UCT) (Kocsis and Szepesvári, 2006) is a popular MCTS algorithm that frames the selection phase as a Multi-Armed Bandit (MAB) problem. A MAB is a sequential decision problem that requires the agent to maximize the total reward obtained from activating one of k arms in each iteration. The reward distribution of the arms is unknown, and the goal is to minimize regret: the difference between optimal and obtained reward. UCT node selection balances between exploration and exploitation by employing the Upper Confidence Bounds (UCB) UCB1 formula.

When the branching factor gets very large, as in the RTS domain, UCT's performance degrades (Ontañón, 2017). The solution proposed by Ontañón (2013) is to formulate the selection phase as a Combinatorial Multi-Armed Bandit (CMAB) problem, most adequate with the combinatorial decision space of an RTS game. In a CMAB, the agent must activate a macro-arm in each cycle, where each macro-arm is a combination of arms, in hopes of maximizing the expected reward.

The proposed NaïveMCTS algorithm uses a naïve sampling strategy, based on a naïve assumption that considers the reward obtained from a macro-arm as the sum of the individual rewards obtained from each underlying arm (Ontañón, 2017).

This assumption and the CMAB formulation allow the decomposition of the problem into $n + 1$ MAB; a global MAB responsible for exploiting macro-arms and n local MABs tasked with exploring the decision space and forming macro-arms. In RTS terms, an arm is a unit-action and a macro-arm is a player-action. NaïveMCTS has shown a satisfying performance when the branching factor grows, but only in small scenarios.

Shleyfman et al. (2014), noted that the macro-arms generated towards the end of the computation budget were not allowed enough exploitation time to produce good estimates, which can mislead search. They proposed to split the computation budget between a candidate generation phase and a candidate evaluation phase. Their two-phase sampling algorithm, Linear Side Information (LSI), is based on a CMAB formulation. In the candidate generation phase, LSI collects side information about each value, which is an estimate of the reward contribution of each value, assumed to be linear. Next, using this information, it generates a set of macro-arms based on either entropy or union. The remaining budget is spent on finding the best macro-arm using sequential halving.

2.3.2 | High-Level Planning (HLP)

Search space dimensionality can be reduced to a more manageable size in order to efficiently find optimal player-actions. Researchers rely on an **abstraction** mechanism, through which, the original search space is replaced by an abstracted one, where each state/action becomes an abstract representation of the original states/actions. The abstraction function is responsible for the abstraction degree and the structure of the abstracted space. HLP consists of searching the abstract space for optimal player-actions. In contrast with LLP methods, HLP methods generally produce better long-term strategic performance and weaker short-term tactical performance (Barriga et al., 2017).

2.3.2.1 | Script-Based Abstraction

The first use of abstraction in an RTS game was done using hard-coded scripts in combat scenarios. Balla and Fern (2009) defined a pair of abstract actions and a unit grouping approach to facilitate search using UCT. Portfolio Greedy Search (PGS), by Churchill and Buro (2013), searches through an abstract action space induced by scripts, using a greedy local-search via hill-climbing. PGS uses a set of combat scripts they call a Portfolio, and iteratively assign to each unit the best-scoring action from a script in the portfolio. The score is derived from a playout (simulation) between scripted opponents, with the outcome calculated using an evaluation function. PGS was implemented in SparCraft (a STARCRAFT combat simulator), and was shown to be highly effective against UCT and ABCD in large combat scenarios (50 vs 50). PGS was later implemented in µRTS, and was capable of playing the full game.

Later, Justesen et al. (2014) used UCT in the space of actions proposed by scripts and further simplified search using a K-means unit clustering. Stanescu et al. (2014) proposed a hierarchical adversarial search algorithm using three abstraction layers, and Uriarte and Ontañón (2014) abstracted the game states using map decomposition into regions, and unit grouping. Wang et al. (2016) described Portfolio Online Evolution (POE), a PGS variant replacing greedy search with a GA.

Moraes and Lelis (2018b) identified a non-convergence issue in PGS, where it may fail to return the best action possible, due to its reliance on a scripted opponent-model during playouts. This way, PGS calculates a best response for that script only, whereas, the actual opponent may play differently. They proposed a Nested Greedy Search (NGS) approach that works in the same way as PGS, with the difference residing in the way NGS handles opponent modelling. NGS greedily searches for the best opponent response to each unit/action combination and calculates optimal actions against it. Moraes and Lelis (2018b) showed that PGS and NGS can be used to play full RTS games. Their NGS implementation in µRTS managed to overcome many state-of-the-art LLP approaches in full game settings, for small to medium map sizes.

Lelis (2017) demonstrated the possibility to reduce the search space dimensionality by partitioning units using a type system. His proposed algorithm, Stratified Strategy Selection (SSS), assigns a script to a set of units judged to belong to the same type. A type system decides on how to partition units using attributes such as current hit points, offensive range and weapon damage. A higher number of types generates finer strategies, and a lower number produces coarser strategies. A proposed variant, SSS+, uses adaptive type systems and a meta-reasoning approach to balance strategy granularity between searches. SSS and SSS+ use a hill-climbing search procedure with a modified evaluation approach. Lelis (2017) tested both SSS variants in different combat scenarios in SparCraft. The results show a substantial advantage over PGS and POE.

Puppet Search (PS) is a high-level search framework proposed by Barriga et al. (2015). PS relies on a number of exposed choice points inserted in predefined, non-deterministic scripts. A script can be a hard-coded, machine-learning based, search based or rule-based agent. A choice point exposes different choices to an overlaying search algorithm, which is expected to decide on the script's next move, or switch to a different script. Search is performed only on the set of choice points exposed, which significantly lowers the branching factor. µRTS's PS version uses the built-in rush scripts implementing a rush strategy[4] with a maximum of 2 choice points in each script. The reported experimentation results using ABCD and UCT are very encouraging, especially in larger maps, compared to benchmark algorithms. Properly designing choice points is an important factor towards a better performance.

MCTS-based planning approaches do not scale well to larger RTS scenarios, where the branching factor gets significantly high. Improving the scalability of NaïveMCTS through action abstraction was proposed by Moraes et al. (2018), and three approaches were presented. In the first approach, Abstraction 1 NaïveMCTS (A1N), naïve sampling samples actions from a subset of abstract actions derived from a set of scripts P instead of the set of low-level actions. In Abstraction 2 NaïveMCTS (A2N), unit-actions are sampled from two distinct sets of scripts, an economy set P_e and a combat set P_b. A1N and A2N significantly outperformed NaïveMCTS in larger scenarios. The third approach, Abstraction 3 NaïveMCTS (A3N), is more related to Hybrid Planning (HyP) and will be described later.

[4]Rush : Continuously train units and immediately dispatch them to attack the opponent's base

2.3.2.2 | Strategy Generation

The main drawback of previous action abstraction approaches is their reliance on a small set of hard-coded strategies, limiting the agent's behavior to a collection of scripted actions. Moreover, dynamic adaptation to the opponent's strategy is not explored enough (Ontañón et al., 2013). All this, can make the agent prone to exploitation.

Strategy Creation via Voting (SCV), proposed by Silva et al. (2018), attempts to solve both problems. It includes a novel-strategy generation mechanism that builds on a small set of hard-coded scripts to generate new strategies, by means of a voting scheme. Additionally, SCV implements a strategy adaptation technique, using a logistic regression model trained from a dataset of matches between different strategies. After every fixed number of decision cycles, it detects the strategy employed by the opponent and adapts its own strategy accordingly.

Finding an optimal abstraction can significantly improve the strategic performance of the agent. Mariño et al. (2018) used an Evolutionary Algorithm (EA) to evolve a set of script-induced abstractions, and find an optimal abstraction. Similarly to SCV, a larger pool of strategies is generated from a smaller set, by varying the parameters of each strategy. Each individual in the EA encodes a subset of strategies that induce a different abstraction. An individual is evaluated by playing a set of RTS matches against all other individuals using either PGS (Churchill and Buro, 2013) or SSS (Lelis, 2017) as a search algorithm in the induced abstract spaces. The optimal individual found was tested against state-of-the-art µRTS agents, and was found to significantly outperform all of them when using SSS.

2.3.2.3 | Machine Learning

Machine learning may be used to learn an abstraction function from expert traces, eliminating the need for domain knowledge. AlphaGo (Silver et al., 2016) achieved a Master-level performance, by means of a Convolutional Neural Network (CNN)-based policy network, trained from a large database of expert Go replays, and improved via a Reinforcement Learning (RL) phase. Later work on AlphaZero (Silver et al., 2018) fully discarded expert knowledge. Inspired by this success, Ontañón (2016) proposed InformedMCTS, an MCTS variant that uses a Bayesian model learned from µRTS replays to inform the selection phase. The author experimented with two distinct models: Calibrated Naïve Bayes (CNB) and Action-type Interdependence Model (AIM), with the latter achieving better performance due to the better representation of the dependency between legal and illegal actions. Yang and Ontañón (2019a) later demonstrated the effectiveness of the C4.5 classifier for such tasks, due to its speed and accuracy.

Reinforcement Learning (RL) over abstracted spaces was proposed by Tavares and Chaimowicz (2018). They modelled the game as a Markov Decision Process (MDP), where the opponent is assumed to be part of the environment, and used a tabular RL approach over options. Options are a set of predefined scripts associated with an abstract state. The RL agent chooses an option in the relevant state, and is rewarded after reaching a different, terminal abstract state. This approach relies on the quality of the script portfolio (options) and the abstraction function.

2.3.2.4 | Domain-Configurable Planners

Planning using domain-configurable formalisms can be regarded as a special case of HLP, since the formalisms are based on expert-provided, high-level domain knowledge. Hierarchical Task Networks (HTNs) are the most utilized domain-configurable planners in µRTS, joined recently by a new research direction using Combinatory Categorial Grammars (CCGs) (Steedman, 2000) as a planning formalism.

An HTN is a hierarchical structure, composed of a goal node and a number of child nodes that represent either primitive or non-primitive tasks. A primitive task can be executed immediately by the agent, whereas a non-primitive task must be decomposed into primitive tasks using a method. A method must have its preconditions satisfied to decompose a non-primitive task. An HTN is fully decomposed when all leaves are primitive tasks. A domain definition determines tasks, methods and ordering constraints.

Adversarial Hierarchical Task Network (AHTN) is an HTN adaptation for adversarial domains, conceived by Ontañón and Buro (2015) for the µRTS domain. It uses an HTN for each player in a game-tree searched through using the MiniMax algorithm. In each iteration, the HTN of the respective player is expanded, and primitive actions get executed. AHTN is adapted for durative and simultaneous actions in the same way as RTMM (Ontañón, 2012). AHTN did not take into account the probability of task failure, and ignored the relationships between tasks. Sun et al. (2017) addressed those issues in AHTN-R, incorporating a task repair mechanism that detects failed tasks, aborts them, and starts an alternative task chosen from a predefined list.

A first attempt to generate plans in µRTS using CCGs (Steedman, 2000) was made by Kantharaju et al. (2018). A CCG is an efficient, linguistically expressive grammar formalism used in Natural Language Processing (NLP) for generating, and recognizing natural language sentences. A CCG separates language-specific structures (lexicon) from grammar. Because of their recognitive and generative properties, CCGs happen to be well suited for recognizing, and generating plans. Kantharaju et al. (2018), were inspired by Geib and Goldman (2011)'s use of CCGs in the context of planning domains. Their proposed agent, µCCG, uses a CCG learned from a set of µRTS plan traces, extracted from replay logs. Lex_{Learn} (Geib and Kantharaju, 2018), the CCG learning algorithm, learns abstract actions from a plan trace, by assigning categories to a series of action types. Using the generated lexicon, planning is done by a similar approach to AHTN, via a MiniMax variant.

Plan recognition (inverse-planning) is the process of determining the goal of an agent, from a series of observed actions. Using CCGs for µRTS plan recognition was proposed by Kantharaju et al. (2019a). They sought to scale CCGs for longer plan traces using a new learning algorithm, Lex_{Greedy}, which reduces the number of learned abstractions by means of a greedy approach. The resulting lexicon was better at recognizing µRTS plans than the one learned by Lex_{Learn}. A Breadth-First Search (BFS) was used for recognition. For larger scenarios, the Lex_{Greedy}-learned lexicon will still contain a very large number of categories per action type, which will impact the performance of BFS recognition. Later, Kantharaju et al. (2019b) scaled-up plan recognition for larger lexicons using MCTS, which is employed to search for the best explanation of the observed series of actions.

2.3.3 | Hybrid Planning (HyP)

HLP approaches usually sacrifice short-term tactical performance for better long-term strategic decision-making. The opposite is true for LLP methods. Hybrid Planning (HyP) performs both types of planning by integrating approaches from both sides into a single method. This integration best imitates the way human RTS players handle strategic and tactical planning, by continuously switching between macro- and micro-management during a match.

To enhance the tactical performance of Puppet Search (PS), Barriga et al. (2017, 2019) sought to combine it with NaïveMCTS. A CNN-based model was trained to predict PS output, and generate strategic decisions. NaïveMCTS would then use the liberated computation budget to improve the tactical decisions of the chosen strategy. The resulting agent, Strategy Tactics (STT), allocates all computation budget to strategic planning if the opponent's units are out of range, and 80% to tactical reasoning otherwise. The reported performance was very promising in large, regular RTS maps, overcoming most of the tested agents.

Strategy Creation via Voting (SCV) (Silva et al., 2018) already relies on a pre-trained model for strategic decisions. Similar to STT, SCV+ (Silva et al., 2018) uses the remaining computation budget to enhance the tactical performance of units in proximity of the opponent's units using ABCD (Churchill et al., 2012). Search is performed in a limited state containing only the relevant units.

Asymmetric abstraction (Moraes and Lelis, 2018a) can be seen as a way to provoke hybrid planning, using a single search procedure. In A3N (Moraes et al., 2018), NaïveMCTS was modified to search in two levels of abstraction. This was done by splitting the set of units into two sets, restricted and unrestricted. An unrestricted unit can acquire any low-level action assignment, whereas, a restricted unit only gets high-level, script-based actions. This way, the agent is allowed finer control over a subset of units. The subset and number of unrestricted units are selected following various strategies, for example, by proximity to enemy units or by remaining hit points.

Neufeld et al. (2019b) proposed a hybrid HTN-MCTS approach using HTN for strategic planning, and NaïveMCTS for tactical planning. Each HTN primitive task is represented by an evaluation function that is passed on to NaïveMCTS. Each evaluation function is conceived to guide NaïveMCTS towards successfully executing a specific task. Thus, the role of NaïveMCTS is to generate actions that optimize the provided function, formulated as a weighted sum of multiple sub-functions (objectives). Function weights were manually tuned. Experimental results show important performance gains against AHTN and NaïveMCTS.

Guided NaïveMCTS (GNS) by Yang and Ontañón (2019b) biases the tree policy of NaïveMCTS by forcing it to select a script-generated action before any low-level action. The goal is to make NaïveMCTS explore at least one high-level action in each iteration, in addition to the low-level actions it usually explores. This way, NaïveMCTS may end-up selecting low-level actions if found better than scripted ones. GNS was found to perform better than NaïveMCTS and the scripts it uses, because it successfully combined the advantages of high- and low-level decisions.

2.3.4 | Partial Observability

In a partially observable setting, some opponent and/or environment information is concealed from the agent, making the problem of online adversarial planning even harder. This is usually a consequence of the fog of war, that is a map overlay that masks all the zones out of the vision range of the player's units, simulating a real warfare situation. Under this setting, scouting and information gathering become crucial. Fog of war is enabled by default in most retail RTS games.

Dealing with partial observability in RTS games did not receive as much attention as the rest of sub-problems (Ontañón et al., 2013; Robertson and Watson, 2014). Nevertheless, some interesting approaches exist, such as the work of Weber et al. (2011), in which they use a particle model to predict opponent units' location in STARCRAFT.

In μRTS, Uriarte and Ontañón (2017) dealt with partial observability in the context of game-tree search by sampling a single believe state from the information set, and using it as a root state for search. They tested multiple believe state generation methods. Yang et al. (2019) proposed a similar approach that deals with discontinuity in unit position history by using a grey-fuzzy approach to minimize prediction error. Antuori and Richoux (2019) proposed a solution to the build order problem under imperfect information, by modeling it as a combinatorial optimization problem and using Rank Dependent Utility (RDU) (Quiggin, 1993) concept to rank different solutions.

2.3.5 | State Evaluation

The value of a player-action is estimated by evaluating the resulting state after n cycles of its execution, through simulation. Evaluation is done by means of an evaluation function. Such function is typically a hand-crafted linear combination of weighted state components, such as unit attributes, resources, ..., etc., that assigns a high value to the advantageous player in the state. RTS baseline evaluation functions include the Lifetime Damage (LTD) function (Kovarsky and Buro, 2005) and Lanchester's laws-of-attrition based function (Lanchester, 1916; Stanescu, 2015). The accuracy and speed of evaluation impacts the performance of planning. Faster evaluation enables more simulations, and consequently, more accurate evaluations. A low evaluation accuracy can mislead search.

AlphaGo's (Silver et al., 2016) use of a CNN-based value network motivated numerous researchers to use similar techniques for RTS state evaluation. Stanescu et al. (2016) proposed a CNN architecture motivated by the lack of spatial information in baseline evaluation functions. Performance improved in multiple search algorithms, but their network was map-size dependent. Barriga et al. (2017) used a Fully Convolutional Network (FCN) that can evaluate arbitrary-sized maps. Yang and Ontañón (2018) sought to learn generalizable, map-independent features, using a CNN architecture separating global and spacial information. CNN-based evaluation methods suffer from a slow evaluation speed, which Huang and Yang (2018) tried to address using a multi-size CNN architecture. Their network features four independent groups of filters of different sizes, intended for parallel computation.

Inspired by HTN planning, Yang et al. (2018) proposed a hierarchical evaluation network composed of three layers of compound and primitive nodes that reflect the current state. The network is decomposed to calculate an evaluation. Neufeld et al. (2019a) proposed to learn optimal evaluation function weights using an Evolutionary Algorithm (EA) for their HTN-MCTS hybrid agent.

Table 2.1: The Top 3 Contenders of Previous µRTS Competitions. Agents with an asterisk (*) in their names are hard-coded scripts.

Year	Tracks (Open and Hidden)	
	Standard	**Partial Observability**
2017	1. STT (Barriga et al., 2017)	1. POLightRush*
	2. POLightRush*	2. BS3NaïveMCTS (Uriarte and Ontañón, 2017)
	3. NaïveMCTS	3. POWorkerRush*
2018	1. Tiamat (Mariño et al., 2018)	1. POAdaptive (Antuori and Richoux, 2019)
	2. UTalcaBot*	2. POLightRush*
	3. Capivara (Moraes et al., 2018)	3. BS3NaïveMCTS

2.4 | RTS AI Competitions

To instigate a higher research interest into RTS AI, annual RTS AI competitions take place alongside a number of respected conferences. The µRTS competition[5] started in 2017 as a side event of the IEEE® CoG (Ontañón et al., 2018). Similar to the StarCraft AI competitions (Churchill et al., 2016), researchers have their agents pitted against each other in a series of tournaments in multiple tracks. µRTS competition includes three distinct tracks: standard track, non-determinism track and partial-observability track.

In each track, a series of round-robin tournaments is performed in each of the predetermined game maps. Maps are organized in two sets, open and hidden, where hidden maps are kept secret from the participants in order to avoid map exploitation. Agents are given a 100*ms* computation budget for each game cycle, and a match is limited by a maximum number of cycles over which it is considered a draw.

The top 3 contenders from the two past tournaments, in all maps, are shown in Table 2.1. Non-determinism track results are omitted for being identical to those of the standard track, except for the 3rd place in 2017's competition, which goes for the built-in POWorkerRush script.

2.5 | Summary

Throughout this chapter, we first described RTS games and defined their general characteristics, and then identified the challenges they pose for both humans and artificial

[5]https://sites.google.com/site/micrortsaicompetition/

agents. We also enumerated the RTS AI levels of abstractions, and formally described RTS games and discussed their complexity, as well as the most common RTS AI research platforms.

Because the focus of our thesis revolves around online adversarial planning techniques, we reviewed and categorized, to our knowledge, all the major game-playing AI techniques conceived in the µRTS domain and its immediate periphery to date. We classified each approach according to the level of abstraction considered during planning, and described the more recent trend of hybrid planning, where strategic and tactical planning are integrated into the same package. This integration led to the emergence of interesting approaches, offering the best compromise between the two levels of planning. We also reviewed some promising planning techniques in partially observable settings, as well as the state-of-the-art in µRTS state evaluation.

The annual µRTS competition results offer some insight about the progress made in the RTS AI domain and its open challenges, through the lenses of µRTS. From those results, and the experiments conducted in the referenced papers, we observe that planning in higher-levels of abstraction consistently yields better performance than in lower-levels, in typical RTS scenarios. This is explained by the presence of expert knowledge in HLP approaches, in the form of scripts, producing better strategic reasoning. Whereas, most LLP approaches forgo expert knowledge. The quality of the script-induced abstractions also contributes to the performance of the approach. 2018's µRTS competition winner, Mariño et al. (2018), managed to find an optimal abstraction that generated better plans. LLP approaches are found to be most useful, when employed for enhancing tactical reasoning, as part of hybrid planning.

Factors limiting the best use of expert knowledge include the absence of true µRTS human-expert replay datasets, and the reliance on a small set of simple built-in scripts. µRTS being a minimalistic research-driven platform, understandably drives away human players, however, we believe this should not stop researchers from finding ways to adapt expert replays from other games to µRTS. In addition, adapting more expert-authored scripts from retail RTS games could prove useful.

Partial observability and non-determinism could use more attention from researchers, since both settings closely reflect real-life scenarios that arise in robotics and control domains. Improving state evaluation by finding an acceptable compromise between speed and accuracy is an interesting line of research, with ample perspectives ahead. Additionally, eliminating the need for a forward-model will expand the application spectrum of planning approaches bound by a forward-model. Lastly, the µRTS platform should be further extended to enable research on more common RTS features, such as unit upgrades, asymmetric factions and multiple resource types. Of course any µRTS extension must adhere to its minimalistic and lightweight philosophy.

In the next chapter we will propose an approach that tries to improve the performance of the state of the art LLP approach, NaïveMCTS, by removing obviously detrimental actions from the search space.

3

Move Pruning in MCTS

"The essence of strategy is choosing what not to do."
— Michael E. Porter

The complexity of RTS games, from an AI perspective, originates from the combinatorial structure of their state and decision spaces. In comparison with classic benchmark games such as Chess or Go, the dimensionality of both state and decision spaces in an RTS game is many orders of magnitude higher (Ontañón et al., 2013). Instead of controlling a single unit in a turn-based fashion, RTS players control multiple units simultaneously in real-time, and usually in a much larger board (map) size. Moreover, the branching factor in an RTS game grows exponentially with the increase in the number of units positioned on the map.

Due to the game's complexity, conceiving a human-challenging RTS game-playing agent is a difficult task to undertake. The predominant approach followed by researchers and practitioners in the domain is to decompose the task into manageable subtasks targeting various degrees of abstraction. Most commonly, an RTS agent combines high-level strategic components and low-level tactical components. Such decomposition is inspired by the way human players interweave micro- and macro-management, and it is shown to be effective by numerous implementations (Barriga et al., 2017; Moraes and Lelis, 2018a; Neufeld et al., 2019b).

Holistic search-based approaches such as MCTS (Browne et al., 2012) enjoyed a remarkable success in computer Go, as demonstrated by AlphaGo (Silver et al., 2016). However, in RTS games, MCTS-based agents struggle with the enormous decision space and fail to scale suitably when the branching factor grows past a certain threshold. Such downside restricts MCTS applicability to limited scenarios, such as tactical planning or small maps. Abstracting the decision space is a tried and tested technique for scaling MCTS-based agents to larger scenarios (Ontañón et al., 2013), at the expense of sacrificing tactical performance due to the coarser actions considered.

We propose an approach to increase the performance and scalability of search-based techniques, particularly MCTS-based ones, by pruning unnecessary and detrimental player-actions from the decision space of an RTS game (Ouessai et al., 2020a). We inspect the low-level structure of the search space and identify detrimental player-actions through domain knowledge. Next, we apply multiple hard-pruning approaches to remove those player-actions during the search. The goal is to reduce the branching factor and explore more promising player-actions. Our approach targets a class of player-actions we identify as Inactive Player-Actions (IPAs) because they tend to keep at least one unit in an inactive state, which can be problematic. The experiments' results using UCT and NaïveMCTS in µRTS show a considerable performance gain relative to the map's size.

3.1 | Monte-Carlo Tree Search (MCTS)

The goal of an RTS game-playing agent is to compute an optimal player-action $a \in A$ at each decision cycle t where the agent can act. MCTS (Browne et al., 2012) is a sampling-based search framework applicable to sequential decision problems, formulated as MDPs (Russell et al., 2010), within large decision spaces unapproachable to systematic search techniques. MCTS estimates the value of actions, sampled using a tree policy, through random simulations. MCTS iteratively constructs a partial game tree following a four-step process at each iteration, as illustrated in Figure 3.1. With each iteration, value estimates of sampled actions would get further refined. The algorithm can be halted anytime to obtain a decision. An MCTS iteration proceeds as follows:

1. **Selection:** Starting from the root node (current state), select a node with unexplored children following a tree policy.
2. **Expansion:** Create and attach a new child node under the selected node.
3. **Simulation:** Start a simulation (playout) from the new node following a playout policy. Usually a default policy is followed, which consists of selecting random actions.
4. **Backpropagation:** Backpropagate the simulation's outcome starting from the new node up to the root node. Nodes along the path will have their visit counts and value estimates updated.

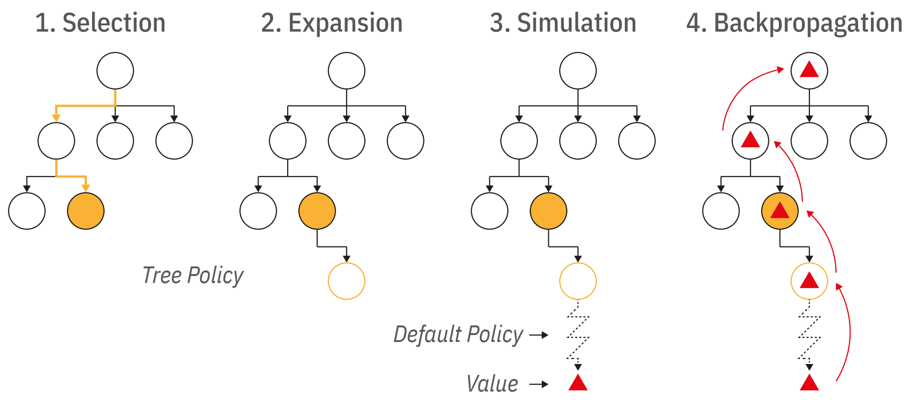


Figure 3.1: The four phases of the Monte-Carlo Tree Search (MCTS). Different MCTS algorithms mainly differ in their tree policy. Nodes represent states, and outbound edges represent actions.

The most visited decision is usually the one returned. Given enough computation budget and a proper exploration/exploitation balance in the tree policy, MCTS is in theory guaranteed to find the MiniMax solution in the limit (Kocsis et al., 2006). Different MCTS algorithms differ principally in the tree policy used in the Selection phase. MCTS can be considered as a RL algorithm, because action reward estimates follow a reinforcement process through each iteration (Sutton and Barto, 2018).

3.1.1 | Upper Confidence bounds for Trees (UCT)

UCT (Kocsis and Szepesvári, 2006) is a popular MCTS algorithm that frames the selection phase as a Multi-Armed Bandit (MAB) (Auer et al., 2002) problem, then uses the UCB1 policy to select nodes for expansion. MABs define a class of sequential decision-making problems, where an agent needs to select an action from K actions possible in order to maximize the cumulative reward obtained. The best course of action is to consistently select the optimal action, however, the underlying reward distributions are unknown. Thus, estimates must be made from previous observations, leading to an exploration/exploitation dilemma. A K-armed bandit can be defined by random variables $X_{i,n}$ for $1 \leq i \leq K$ and $n \geq 1$, with i identifying the arm (action chosen) of the bandit. A MAB must be approached through a policy that aims to minimize the agent's regret (Browne et al., 2012). Cumulative regret after T iterations is defined as:

$$R_T = \sum_{t=1}^{T}(\mu^* - \mu_t) \tag{3.1}$$

Where μ^* represents the maximum expected reward, and μ_t the reward obtained by the action selected at iteration t. Regret thus represents the difference between the ideal choice and the choices made, over T iterations. Auer et al. (2002) proposed to compute a Upper Confidence Bounds (UCB) value that determines whether an action will be optimal. The UCB1 policy used by UCT requires the selection of an arm (action) that maximizes the following:

$$UCB1 = \overline{X_j} + c\sqrt{\frac{\ln n}{n_j}} \tag{3.2}$$

Where $\overline{X_j}$ is the average reward obtained from arm j. n_j represents the number of times arm j was selected and n is the total number of selections. The first term ($\overline{X_j}$) pushes for the selection of the best arm (exploitation), whereas the second term increases the chances of selecting less explored arms (exploration). An exploration factor c is added to the second term to control exploration intensity.

UCT works well in high branching-factor domains, such as Go, but suffers greatly when the decision space has also a combinatorial structure, as in RTS games. This drawback is due to UCB1's implicit requirement to explore all possible actions at least once to commence exploitation. The short decision cycle and huge average number of possible moves at a decision point in RTS games do not allow UCT the chance to explore all moves.

3.1.2 | NaïveMCTS

The idea of NaïveMCTS is to replace the MAB-based tree policy in UCT with a more fitting Combinatorial Multi-Armed Bandit (CMAB)-based policy. CMABs are a natural match for RTS games due to their support for combinatorial decision spaces. Instead of sampling a single variable as in a MAB, A CMAB permits sampling a combination of multiple variables. A CMAB is defined by the following (Ontañón, 2013):

- A set of n variables $X = \{X_1, \ldots, X_n\}$, where each variable X_i may take K_i different values $\mathcal{X}_i = \{v_i^1, \ldots v_i^{K_i}\}$
- A reward distribution $R : \mathcal{X}_1 \times \ldots \times \mathcal{X}_n \rightarrow \mathbb{R}$ depending on value of each variable.
- A legality function $V : \mathcal{X}_1 \times \ldots \times \mathcal{X}_n \rightarrow \{true, false\}$ that checks whether a value combination is legal or not.

A sampling strategy for a CMAB needs to find a legal combination of values, known as a macro-arm, that yields maximum rewards. If $v_1^*, \ldots, v_n^*$ represents the values that grant the maximum expected reward μ^*, the regret ρ_T of a CMAB sampling strategy after T iterations is defined as:

$$\rho_T = T\mu^* - \sum_{t=1}^{T} R(x_1^t, \ldots, x_n^t) \tag{3.3}$$

Where, $x_1^t, \ldots, x_n^t$ represent the macro-arm selected by the sampling strategy at iteration t. NaïveMCTS exploits the underlying structure of RTS decision spaces using a *naïve sampling* approach based on a *naïve assumption* that considers the reward estimate of a macro-arm as the sum of the reward estimates obtained by each underlying arm (single variable value). Thus, the reward distribution is decomposed as such:

$$R(x_1, \ldots, x_n) = \sum_{i=1}^{n} R_i(x_i) \tag{3.4}$$

The naïve assumption makes it possible to break down the CMAB problem into $n+1$ MAB problems:

- **Global MAB** (MAB_g)**:** Considers the whole CMAB problem as a MAB problem, thus, each sampled legal macro-arm is treated as an arm. The global MAB is initially empty, and will be filled with sampled macro-arms in each subsequent iteration.
- **Local MABs** (MAB_i)**:** For each variable $X_i \in X$, a MAB_i, specific to X_i is defined. The job of local MABs is to form macro-arms by sampling the values of each variable separately. The macro-arms are added to the global MAB.

Local MABs use the naïve assumption to explore different value combinations that may yield a high reward. The global MAB would then exploit the combinations that resulted in the highest reward. At each iteration, a policy π_0 decides whether to explore or exploit. On explore, a macro-arm $x_1^t, \ldots, x_n^t$ is sampled using a policy π_l to independently select a value for each variable. On exploit, a macro-arm $x_1^t, \ldots, x_n^t$ is sampled using a policy π_g which uses the sampled combinations of the global MAB.

NaïveMCTS uses ϵ-greedy policies for π_0, π_l, and π_g, which take parameters ϵ_0, ϵ_l, and ϵ_g, respectively. An ϵ-greedy policy uses a random policy with a probability ϵ, and a greedy policy with a probability $1-\epsilon$.

3.2 | Related Works

Dealing with the enormous RTS decision space in the context of MCTS is an open problem continuously receiving contributions. By treating the selection phase as a CMAB, NaïveMCTS effectively adapts MCTS to combinatorial search spaces. Nevertheless, the decision space remains the same, and the algorithm still suffers from high dimensionality. Downsizing the search space's dimensionality is usually done through action abstraction as detailed in section 2.3.2 and 2.3.3 of Chapter 2. Abstraction is effective in reducing the overall branching factor, but move-pruning (Marsland, 1986) could also play a significant role towards the same end.

If we adjust our perspective, HLP methods can be regarded as indirect move pruning approaches (Yang and Ontañón, 2020). By focusing on a set of promising expert-based player-actions, these approaches effectively prune the search space of all the remaining player-actions, significantly reducing the branching factor. Still, such practice can also become unsafe and prone to exploitation, due to the coarser player-actions considered, resulting in a loss of tactical performance. To address this issue, HyP approaches combine low- and high-level searches (Barriga et al., 2017; Moraes et al., 2018; Neufeld et al., 2019b).

Directly pruning the player-actions responsible for weak performance can be an alternative approach towards focusing the search on promising actions, without compromising tactical strength. In the context of Chess (Heinz, 1999) and Shogi (Hoki and Muramatsu, 2012), several forward pruning methods (Lim and Lee, 2006) such as Null-move pruning and futility pruning were utilized to reduce the branching factor and enhance AlphaBeta search. In Go, a domain-dependent pruning approach was implemented in UCT (Huang et al., 2010), exploiting territory information. Similar MCTS improvements were applied in the games of Hex (Arneson et al., 2010), Havannah (Duguépéroux et al., 2016), and DeadEnd (He et al., 2008). In digital games, Sephton et al. (2014) enhanced MCTS by applying a knowledge-based move pruning approach for the strategic card game, LORDS OF WAR.

In this chapter, we suggest using a domain knowledge-based hard-pruning technique for MCTS agents in RTS games. This technique targets a specific type of player-actions prevalent in all RTS games. To our knowledge, this is the first application of a move-pruning approach in the context of RTS games.

3.3 | Move Pruning

We propose to act directly on the decision space and hard-prune a subset of decisions we deem irrelevant and/or detrimental to the performance of MCTS. By doing so, MCTS will be freed from sampling those decisions and simulating their outcomes. The recovered computation time will be spent on exploring more relevant and significant decisions, which would improve the playing strength and scalability of MCTS.

Table 3.1: The unit-action types available for each unit-type in µRTS.

Units	Actions					
	Move	Attack	Harvest	Return	Produce	Wait
Worker	●	●	●	●	●	●
Light	●	●				●
Ranged	●	●				●
Heavy	●	●				●
Base					●	●
Barracks					●	●

As a first attempt, we chose to focus on player-actions having the highest chance of misleading search and negatively impacting the playing strength. Out of these player-actions, we believe Inactive Player-Actions (IPAs) naturally come first. Thus, we implemented several pruning approaches that keep a predefined number (fixed or relative) of those player-actions and prune the remaining. We will briefly discuss the structure of RTS player-actions next and then define IPAs.

3.3.1 | Unit-Actions and Player-Actions

In a typical RTS game, each unit type can execute a distinct set of actions known as unit-actions. Table 3.1 enumerates the unit-action types executable by each unit-type in µRTS. The Worker unit-type is the most versatile, followed by assault units (Light, Ranged and Heavy) and structures (Base and Barracks). The attributes of a unit-type define the effect of its unit-actions. For instance, the damage attribute controls how much damage a unit-type causes when executing the Attack unit-action. Thus, even for common unit-actions, each unit-type may behave differently. All unit-action types, except Wait, require an argument that determines the target of the action. The Wait unit-action type requires a numeric argument specifying the number of cycles ahead at which the unit must remain inactive. Wait is the only unit-action type unaffected by unit attributes and executable by all unit-types.

A player-action $a \in A$ issued to n units at a given game cycle can be regarded as a tuple, $a = (\alpha_1, \alpha_2, \ldots, \alpha_n)$, where each component α_i is a unit-action issued to the i-th unit. Given the average number of legal unit-actions available to each unit, m, the number of all possible player-actions or the branching factor, b, can be estimated as $b = m^n$. We seek to lower b by finding ways to decrease m without negatively impacting the playing strength.

3.3.2 | Inactive Player-Actions (IPAs)

We define an IPA as a player-action having at least one Wait (inactive, idle, or no-op) unit-action as a component. Being the most prevalent non-critical unit-action, Wait unit-actions make for a good pruning target. The Wait unit-action is continuously available to all units, regardless of their situations. Thus, it strongly contributes to the inflation of the search space. Nonetheless, Wait unit-actions can be advantageous for a unit, usually in the following situations:

- **Trapped unit:** No active unit-action is possible. The unit is caught in a situation where all possible unit-actions are illegal. Waiting for a predefined duration is the only option to choose in hopes the situation is resolved.
- **Tactical waiting:** The unit anticipates for a chance to execute a high-value unit-action. Here, the unit expects a suboptimal action by an opponent unit (via lookahead) and chooses to Wait in anticipation for it. Executing the high-value action happens afterward. This behavior is frequently observed in tactical skirmishes.

Although potentially useful, Wait unit-actions can also have a devastating effect on the playing strength if improperly chosen. According to our observations, it is not unlikely for a search-based agent (MCTS or otherwise) to assign a Wait unit-action to a unit in a situation where better options exist. In such cases, doing nothing can be the worst decision possible. We identify three disadvantageous situations where Waiting cannot be a sound decision:

- **Waiting in front of opportunity:** Here, the unit can seize an immediate opportunity, such as Harvest resources, Return harvested resources, or safely remove an opponent unit. Instead, the unit is assigned Wait.
- **Waiting in the face of danger:** The unit is facing an immediate danger and holds the necessary options to avoid it, but instead, it is assigned a Wait unit-action.
- **Waiting frequently:** The unit is assigned Wait unit-actions more often than the other unit-actions, in the absence of immediate dangers/opportunities, making it less effective in pursuing opportunities and almost passive.

The presence of one Wait unit-action in a player-action (thus, IPA) is enough to introduce a risk of encountering one of the disadvantageous situations. The more Wait unit-actions in an IPA, the higher this risk gets. Thus, we believe that pruning the majority of IPAs from the search space, while preserving a fraction as a safety measure, to account for trapped units and tactical waiting, can be beneficial to MCTS.

3.3.3 | Pruning Techniques

The radical pruning approach would be to remove all IPAs from the search space, basically removing the Wait unit-action from the set of unit-actions of all unit types. Thus, diminishing m by 1 and obtaining a branching factor $b' = (m-1)^n$, which represents a considerable decline from b. As an example, if we have $m = 5$ unit-actions on average in a given game state with $n = 6$ units, then $b \approx 1.56 \times 10^4$ and $b' \approx 4 \times 10^3$. The total number of IPA removed would be: $v = b - b' = 1.16 \times 10^4$. The reduction is significant, but we intend to keep a portion of IPAs to deal with trapped units and tactical waiting.

Detecting trapped units is a simple task. But dealing with tactical waiting can be elusive since there is no simple way to differentiate between waiting as a tactical choice, and waiting as a bad decision until witnessing the consequences. Random playouts do not offer a reliable answer in that regard. We propose four pruning approaches that capture IPAs and decide whether to allow or prune them according to a given parameter. These approaches preserve all IPAs involving trapped units and allow a predetermined number/rate of random IPAs in hopes of preserving tactical waiting situations. The remaining IPAs are all considered disadvantageous and are systematically hard-pruned. We do not re-insert pruned IPAs because we consider the non-pruned player-actions more urgent to explore. The pruning approaches are described as follows:

- **Random Inactivity Pruning - Fixed (RIP-F(k)):** Allow a fixed number k of IPAs.
- **Random Inactivity Pruning - Relative (RIP-R(p)):** Allow a percentage of IPAs p relative to the total number of removable IPAs.
- **Dynamic Random Inactivity Pruning - Fixed (DRIP-F(k_1, k_2)):** Allow k_1 IPAs when the agent's units outnumber the opponent's units, and k_2 IPAs otherwise.
- **Dynamic Random Inactivity Pruning - Relative (DRIP-R(p_1, p_2)):** Allow p_1 percent of IPAs when the agent's units outnumber the opponent's units and p_2 percent of IPAs otherwise.

The intuition behind dynamic approaches is to equalize the chances of performing tactical waiting when the agent does not hold a numerical advantage. This is done by allowing more IPAs when the agent is outnumbered ($k_2 > k_1$ or $p_2 > p_1$).

Algorithm 1 The general IPA pruning algorithm.

```
function PRUNE($a$)                                   ▷ $a$ : A sampled player-action
    if ISIPA($a$) then
        if TRAPPEDUNIT($a$) then return $a$
        end if
        if $a \in prunedIPAs$ or CHECKPARAM() then
            repeat
                $prunedIPAs.addIfNotExist(a)$
                $a \leftarrow$ SAMPLEACTION(gameState)
            until ISIPA($a$) == false
        end if
    end if
    return $a$
end function
```

These approaches can be easily implemented as part of any search algorithm, as shown in Algorithm 1. Each time a player-action a gets sampled, PRUNE(a) is called to decide whether to keep or replace a, in case it is an IPA. If a is a Non-IPA or an IPA involving a trapped unit, it will be returned as-is (lines 3 and 12). ISIPA(a) returns *true* if a has at least one Wait unit-action, and TRAPPEDUNIT(a) returns *true* if a Wait unit-action in a belongs to a trapped unit. The algorithm keeps track of previously-pruned IPAs in the *prunedIPAs* list to prevent re-insertions. The conditional expression in line 5 defines the pruning condition, which is satisfied either if a was previously pruned, or if CHECKPARAM() returns *true*.

The pruning approaches differ in the implementation of CHECKPARAM(). For instance, in RIP-F(k), CHECKPARAM() returns *true* only if the number of IPAs allowed is higher than k. The loop in lines 6-9 will keep re-sampling for new player-actions until a is replaced with a non-IPA. Finally, a new non-IPA or an IPA allowed by CHECKPARAM() is returned.

3.4 | Experiments & Results

To study the effect of pruning IPAs on MCTS, we implemented the four aforementioned pruning techniques in UCT and NaïveMCTS and conducted various experiments in µRTS. Indeed, UCT's performance suffers greatly in RTS scenarios due to UCB1's limitations in combinatorial search spaces (Ontañón, 2017), nevertheless we wanted to test if pruning IPAs would alleviate the dimensionality burden and results in performance improvement. Integrating IPA pruning in UCT and NaïveMCTS generated new agents that we refer to by suffixing the acronym of the technique to that of the original search approach. For instance, the agent using RIP-R(p) with UCT or NaïveMCTS is noted as UCT-RIP-R(p) or NMCTS-RIP-R(p).

We first analyzed the performance of RIP-F(k) and RIP-R(p) relative to the number of IPAs allowed, the map's size, and the MCTS algorithm in use. We then took the top-performing pruning approaches for each MCTS algorithm and map size and performed a round-robin tournament with other µRTS agents. Afterward, we examined the impact on the branching factor and performed a scalability test in larger maps. The experiments were carried out on two PCs with Intel® Core® i5 and i7 CPUs, clocked at 3.1Ghz and 3.4Ghz, respectively, using the latest version of µRTS as of the 30th of March 2020.

3.4.1 | Pruning Analysis

To analyze the influence of IPA pruning on the performance of MCTS, we ran a series of experiments involving each MCTS agent and non-dynamic IPA pruning approach. We defined two distinct sets, F and R, composed of a selection of values that can be taken by the parameters of RIP-F(k) and RIP-R(p), respectively. Next, we ran 500 matches (switching sides after 250 matches) between the MCTS agent enhanced with an IPA pruning approach, and the non-pruning version of the same MCTS agent for each respective value in F or R. The process was repeated for each *basesWorkers* map of size 8×8, 12×12 and 16×16. We define F and R as follows:

- $F = \{0, 1, 5, 10, 50, 100, 500, 1000, 5000, 10000\}$
- $R = \{0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1\}$

The total number of matches played for a single MCTS agent amounts to $(500 \times |F| \times 3) + (500 \times |R| \times 3)$, yielding 63000 matches for both UCT and NaïveMCTS. In all experiments, we kept the default UCT and NaïveMCTS parameters as defined in the µRTS codebase, for all variants. Agents were given $100ms$ per frame as a computation budget. The experiment results are expressed in Figure 3.2's plots.

Overall, we can see that IPA pruning is responsible for a performance gain of variable rates, relative to the number of IPAs allowed and the branching factor represented by the map's size. In UCT-RIP-F(k), allowing a few IPAs $(0 < k \leq 5)$ significantly increases UCT's performance. However, the more IPAs are allowed, the more performance decreases until pruning losses its effect $(k \geq 1000)$. The same trend is witnessed in UCT-RIP-R(p). Allowing a small percent of IPAs $(0.1 \leq p \leq 0.3)$ increases UCT's

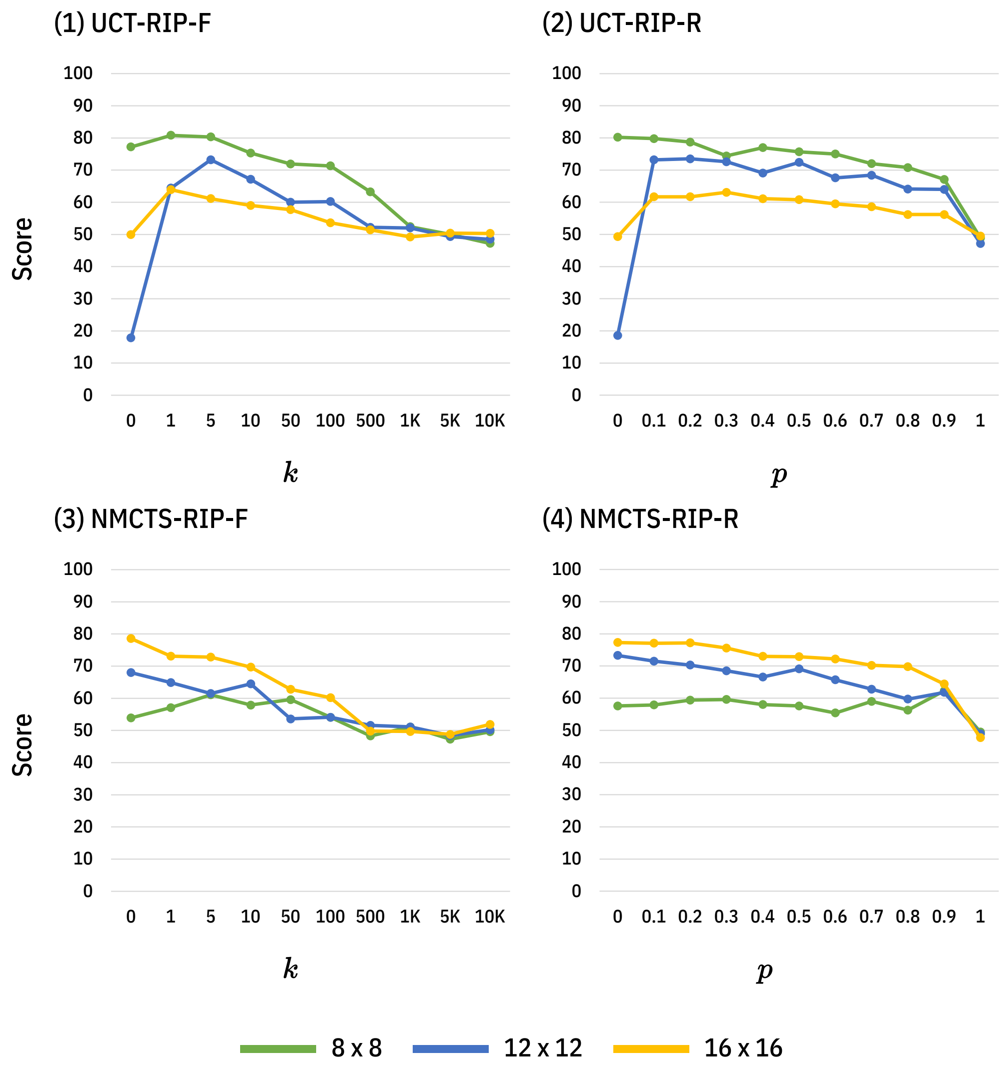


Figure 3.2: Results of the pruning analysis experiments. Each data point represents 500 matches between a basic MCTS agent and the same agent enhanced with IPA pruning. The vertical axes represent the score obtained by the latter agent. The score was calculated as such: $score = ((Wins + (Draws/2))/500) \times 100$. A match is considered a draw if no winner has been decided after 3000, 3500, and 4000 cycles in each map of size 8×8, 12×12, and 16×16, respectively.

performance, but the more we raise p the more performance drops. We note that UCT-RIP-R($p = 1$) and UCT-RIP-F($k \geq 1000$) are equivalent to non-pruning UCT.

The highest performance gain was recorded in the smallest 8×8 map with the lowest branching factor, and the lowest gain was recorded in the largest 16×16 map. This is expected from UCT since its sampling strategy (UCB1) is ineffective in combinatorial search spaces. Thus, pruning IPAs is not enough to scale UCT's performance. Pruning all IPAs ($k = 0$ or $p = 0$) hurts UCT's performance in larger maps due to the large number of IPAs encountered. This causes frequent player-action re-samplings that rapidly consume the computation budget, leaving very little time to exploitation. Although this effect is offset in the 16×16 map by the large number of draws, it is quite clear in the 12×12 map.

From the perspective of NaïveMCTS, IPA pruning exhibits the same effect as in UCT, with two key differences. First, in the smallest 8×8 map, NaïveMCTS performs optimally when more IPAs are allowed (in comparison with UCT), that is, when $5 \leq k \leq 50$ in NMCTS-RIP-F(k) and $p = 0.9$ in NMCTS-RIP-R(p). This is probably because naïve sampling already handles small scenarios well and can gain an advantage if a portion of IPAs is kept to explore tactical waiting situations. However, in larger maps, pruning all IPAs ($k = 0$, or $p = 0$) yields the highest performance gain. Here, due to the bigger branching factor, pruning IPAs significantly contributes to the better utilization of the computation budget.

The second difference with respect to UCT is the scalability of performance, relative to the increase in the branching factor. As opposed to UCT, NaïveMCTS enhanced with IPA pruning delivers its best performance in the largest 16×16 map, followed by the 12×12 map. Having fewer IPAs in the search space seems to allow naïve sampling to sample more interesting player-actions, instead of wasting time on IPAs. Moreover, pruning IPAs increases the movement frequency of units, resulting in an enhanced ability to explore large maps. We will see further how this translates to larger maps.

3.4.2 | Best Pruning Approaches

Concerning dynamic pruning approaches, DRIP-F(k_1, k_2) and DRIP-R(p_1, p_2), we have conducted a similar experiment using UCT, by fixing k_1 (or p_1) to the optimal k (or p) value found for each map size and performing 500 matches for each k_2 (or p_2) value from F (or R). We omitted the results because of space constraints. For NaïveMCTS, dynamic pruning did not bring any improvement over non-dynamic approaches, based on preliminary tests. Thus, performing extensive experiments was not necessary. The best performing IPA pruning approaches for each map-size and MCTS algorithm are shown in Table 3.2.

For UCT, dynamic approaches work best in the small and medium-sized maps due to the frequent encounters between opposing units. In the largest map, exploration becomes more urgent, rendering the dynamic approaches ineffective.

Table 3.2: The best performing pruning approaches, by map size and search algorithm.

	8×8		12×12		16×16	
	Approach	Score	Approach	Score	Approach	Score
UCT	DRIP-F$(1, 0)$	82.2	DRIP-R$(0.2, 0.4)$	76.8	RIP-F(1)	63.9
NaïveMCTS	RIP-R(0.9)	62.3	RIP-R(0)	73.3	RIP-F(0)	78.6

3.4.3 | Performance Analysis

To further assess the performance impact of IPA pruning on MCTS agents, we have run a round-robin tournament between eight µRTS agents, including two IPA pruning MCTS agents, under µRTS competition conditions. The tournament consists of 100 iterations, where in each iteration, every agent plays a match against the other agents resulting in $8 \times 7 \times 100 = 5600$ matches in each of the maps used previously. The participating µRTS agents include four baseline agents and one top performing agent from 2019's µRTS competition, MixedBot:

- **NaïveMCTS:** The original unmodified NaïveMCTS.
- **RandomBiased:** Selects actions randomly, with a bias towards attacking and harvesting. Built-in baseline agent.
- **POWorkerRush:** Continuously produces workers and sends them to attack the opponent. Built-in baseline agent.
- **POLightRush:** Same as the above, but using Light units.
- **MixedBot:** Relies on two separate agents; Tiamat (Mariño et al., 2018) for strategic decisions and Capivara (Moraes et al., 2018) for tactical decisions. Both based on search space abstraction through scripts, and both tuned for different map sizes.

In addition to the following IPA pruning agents:

- **NMCTS-Random Inactivity Pruning (RIP):** NaïveMCTS integrating the best performing IPA pruning approach for each map size, as defined in Table 3.2.
- **UCT-RIP:** Same as the above, but based on UCT.

We also included unmodified UCT for the sake of comparison. The global tournament results are reported in Table 3.3, and the results by map-size are shown in Figure 3.3. The score is calculated similarly to the previous experiment.

Table 3.3: Overall tournament results (Row agent vs. column agent)

	POWorkerRush	MixedBot	NMCTS-RIP	POLightRush	NaïveMCTS	UCT-RIP	UCT	RandomBiased	Average
POWorkerRush		74.5	60.6	100	69.8	72.3	89.5	100	81
MixedBot	30.5		74.8	82	89.6	86.8	99	100	80.4
NMCTS-RIP	42.1	26		47	72.1	80.6	96	100	66.3
POLightRush	0	30.8	58		66.3	65.3	66.6	98.3	55
NaïveMCTS	34.5	9.3	31	36.3		69.3	94	100	53.5
UCT-RIP	26.8	14.1	20.8	35.5	30.6		78	98.1	43.4
UCT	13	1	2.3	34.6	7.5	21.3		84.8	23.5
RandomBiased	0	0	0	6.6	0	1	13		2.9
Average	21	22.2	35.3	48.9	48	56.7	76.6	97.3	

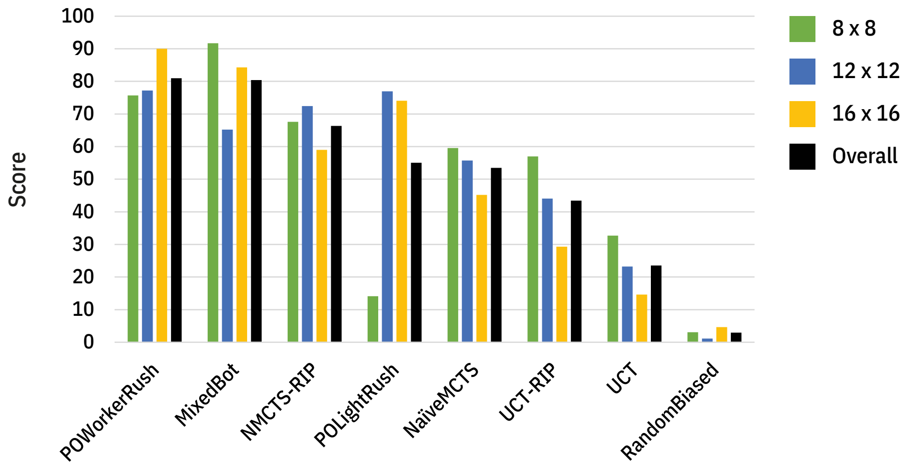


Figure 3.3: Results of the tournament in each map.

The results demonstrate how IPA pruning positively affects the playing strength of UCT and NaïveMCTS. Looking at the average scores, NMCTS-RIP achieved a 12.8 points increase with respect to NaïveMCTS, and UCT-RIP achieved a near 20 points increase, relative to UCT. IPA pruning in UCT (UCT-RIP) managed to shrink the performance gap between UCT and NaïveMCTS from 30 points to 10.1 points, noticeable in Figure 3.3 where UCT-RIP's performance closely matches that of NaïveMCTS in the 8×8 map. Moreover, NMCTS-RIP was able to score a higher average than POLightRush, one of the strongest scripts usually outranking NaïveMCTS. Against each agent, both NMCTS-RIP and UCT-RIP obtained significantly higher scores than those of NaïveMCTS and UCT respectively.

As expected, scripts and script-based approaches exhibit a superior performance versus low-level MCTS search approaches, due to the presence of expert knowledge in the form of hard-coded scripts. Expert knowledge helps in avoiding detrimental player-actions by focusing the search on a limited set of player-actions judged more rewarding. However, this comes at the cost of lower decision granularity and higher exploitation risk. By pruning detrimental player-actions, we hope to focus the search on a wider range of interesting player-actions and keep a higher degree of decision granularity. The fact that NMCTS-RIP could achieve a higher average score than MixedBot in the 12×12 map signifies that our approach could be promising.

Under the µRTS competition settings, both holistic MCTS agents and those that rely on MCTS only for low-level tactical planning, e.g. (Barriga et al., 2017) and (Moraes et al., 2018), would benefit from IPA pruning and see a tactical performance gain that positively impacts their overall performance.

3.4.4 | Branching Factor & Scalability

To better grasp how IPA pruning affects MCTS performance, we took 100 mid-game states from matches between NMCTS-RIP and NaïveMCTS and ran a 100ms search, starting from those states for both agents. The search was limited to a single ply only ($maxDepth = 1$), and the mid-game state was defined at 400, 600 and 1000 game cycles for each map of size 8×8, 12×12 and 16×16, respectively. The statistics collected during these searches are reported in Table 3.4.

Table 3.4: Branching factor, sampled actions, and IPA pruning statistics

Agent	NMCTS-RIP			NaïveMCTS		
Map Size	8×8	12×12	16×16	8×8	12×12	16×16
Avg. Branching Factor, b (Incl. IPAs)	1782	2.48×10^{7}	9.52×10^{11}	784	6.51×10^{6}	1.84×10^{11}
Avg. Unit Count, n	5.19	14.39	20.92	5.32	14.39	20.06
Avg. Unit Actions, m (Incl. Wait)	3.15	2.67	3.11	2.71	2.54	3.05
Avg. Branching Factor, b' (w/o IPAs)	54	1561	6.12×10^{6}	17	494	1.87×10^{6}
Avg. Sampled Actions	70.81	75.27	71.25	71.76	80.11	76.85
IPAs Rate in Sampled Actions	33.82%	71.81%	76.72%	80.03%	99.24%	99.88%
Pruned IPAs Rate (WRT. Sampled Actions)	6.04%	13.32%	24.56%	0%	0%	0%

We can see that the branching factor b in mid-game states is higher in NMCTS-RIP, as a result of the similarly higher unit-actions average m. This, in turn, is the consequence of the units being spread out on the map due to lesser IPAs (more movements),

Table 3.5: NMCTS-RIP-F(0) results versus NaïveMCTS in larger maps.

Map Size	Wins	Losses	Draws	Score
24×24	52	6	42	73
32×32	43	1	56	71

leading to more space between units, and more possible actions for each. NMCTS-RIP samples player-actions from a subset of the decision space having a lower bound branching factor b', expanded by the number of IPAs involving trapped units, and a random set of IPAs allowed by the pruning approach.

The rate of IPAs in both branching factor and sampled actions grows proportionally to the branching factor, as expected. The rate of IPAs in sampled actions is significantly high in NaïveMCTS, reaching near 100% in larger maps. Whereas in NMCTS-RIP, this rate drops under 34% and would not go beyond 77% in the tested maps. Moreover, the rate of pruned IPAs increases proportionally with the branching factor. Therefore, we may conclude that in larger maps NaïveMCTS gets fully overwhelmed by IPAs, while NMCTS-RIP prunes more IPAs and focuses on a larger number of possibly better player-actions. This further highlights the detrimental effect of the overabundance of IPAs.

We have run 100 matches (switching sides after 50 matches) between NMCTS-RIP-F(0) and NaïveMCTS in larger 24×24 and 32×32 maps, to test the performance scalability of IPA pruning in larger scenarios. The results in Table 3.5 suggest a stable score trend and an increasing win/loss ratio proportional to the map's size. Additional experiments in these scenarios are planned in the context of our next works.

3.5 | Summary

In this chapter, we studied the possibility of employing move pruning as a way to enhance MCTS performance in the context of RTS games. We have identified a class of player-actions that can negatively impact the performance of low-level MCTS approaches. We labeled those actions as Inactive Player-Actions (IPAs) due to their tendency to keep at least one unit in an inactive state. Several pruning approaches were conceived to prune IPAs, taking into account the existence of possibly useful IPAs. We then carried out a range of experiments to test the validity of our approaches and discussed the obtained results. According to the results, pruning IPAs is associated with a meaningful performance gain, due to the reduced branching factor and the increased focus on more interesting player-actions. IPA pruning in NaïveMCTS has demonstrated an impressive performance across increasingly larger maps, especially when all excessive IPAs get pruned. Therefore, we conclude that NaïveMCTS can safely ignore all superfluous IPAs in such situations, which will grant a risk-free performance boost in an RTS game (Ouessai et al., 2020a).

Move pruning in this context can be seen as *inverse* action abstraction, since we are trying to find the set of player-actions to avoid, whereas, action abstraction methods seek to find the set of player-actions to focus on exclusively. Pruning low-quality player-actions could result in a more flexible and granular decision space, rather than the coarser space induced by action abstractions (scripts).

µRTS agents integrating a low-level search technique with IPA pruning could gain an improved tactical reasoning ability. This improvement would positively influence the agent's performance in the µRTS competition.

Researching more prunable player-action types falls into the scope of our next work, along with further analysis of IPA pruning in larger scenarios, and the analysis of the impact of IPA pruning in multi-level search approaches such as STT (Barriga et al., 2017) and A3N (Moraes et al., 2018). We believe that further research into the low-level structure of the RTS decision space could lead to a deeper understanding of the general features of higher-quality decisions.

In the following chapter, we propose an integrated action and state abstraction framework that relies on RTS heuristics to reduce the size of the decision space. In this framework, it is also possible to make use of IPA pruning as post-processing phase.

4 Parametric Action Preselection

"Strategy without tactics is the slowest route to victory.
Tactics without strategy is the noise before defeat."
— Sun Tzu

Because RTS games possess a state space and a decision space of massive proportions, artificial agents often need the help of domain knowledge to navigate the vast array of possibilities. Recently, the popular RTS game STARCRAFT II saw the rise of the breakthrough agent, AlphaStar (Vinyals et al., 2019), to the rank of Grandmaster. Conceived by DeepMind, AlphaStar uses domain knowledge in the form of match replays from top human experts for a supervised learning phase, in addition to a multitude of machine learning techniques, including self-play. Nonetheless, the tremendous computational effort required in the training phase, paired with access to custom hardware, reduces the chances of reproducibility under downscaled typical conditions.

Monte-Carlo Tree Search (MCTS)-based agents could be a promising alternative due in part to their flexible computational needs and their proven, successful track record in games with large decision spaces such as Go and its kin. However, MCTS applicability in RTS games is still limited, given its forward-model requirement (simulator needed to quickly simulate the outcomes of actions), and expectedly, because of its struggle against expansive, combinatorial decision spaces. Several research evidences suggest that guiding MCTS by means of expert knowledge may contribute to its competitiveness, depending on how the expert knowledge may be integrated in the search process.

In the previous chapter we explored how a knowledge-based move pruning approach could improve MCTS' performance. Raw domain knowledge was used to design a player-action sampling scheme that discards a proportion of IPAs, known for their detrimental qualities (Ouessai et al., 2020a). As mentioned in Chapter 2, some

HLP and HyP approaches use domain knowledge in the form of scripts. A script is a set of rigid rules, designed by a domain expert, that govern the behavior of an agent. In this chapter, we propose an action preselection algorithm that relies on the building blocks of scripts rather than full scripts to guide action selection (Ouessai et al., 2020b).

Expert-authored scripts combine several smaller scripts, we call heuristics, to form a scripted agent. A heuristic represents an isolated task performed by a unit or a group of units, such as harvesting or attacking. Our algorithm depends on parametric heuristics to generate a wide variety of scripts that can be used to preselect actions for predefined groups of units, to feed into an MCTS agent. The heuristics' parameters can be modified in real-time to adjust both the decision granularity and the adopted strategy, which opens the perspectives for dynamic strategy adaptation through MCTS. Experimentation results in the μRTS test-bed reveals a significant MCTS performance gain related to the reduced branching factor and the more focused decision space.

4.1 | Related Works

The first bottleneck encountered by an algorithm searching for an optimal decision in an RTS match is the very high dimensionality of the decision and state spaces, exacerbated by the real-time constraints. Search approaches would easily get lost while looking for optimal actions and may settle for suboptimal actions, in almost every instance. As indicated in Chapter 2, the most prevalent solution to this dimensionality curse assumes the form of an intermediate abstraction layer that conceals low-level representations in favor of higher-level ones, deemed more valuable by expert knowledge. As commonly practiced, an abstraction layer in the decision space is either implemented through a portfolio of expert scripts or a probability distribution over expert actions. On the other hand, an abstraction layer in the state space results from a manual or an automatic clustering process.

All the approaches enumerated under Section 2.3.2, and 2.3.3 of Chapter 2 can be considered related to the proposed approach, since all share the same trait of considering abstracted decision and/or state spaces. One key characteristic distinguishing our approach from the other HLP and HyP methods, is its reliance on parametric heuristics to manipulate and shape the abstraction layer, which opens up many interesting possibilities for RTS game-playing agents.

4.1.1 | Relation with Dynamic Scripting

Action preselection could be seen as analogous to dynamic scripting (Spronck et al., 2006). This point of view is justified because both methods combine small-scale scripts (Heuristics vs. rules) to form full-scale scripts or a strategy. The primary goal of dynamic scripting is to automatically adapt a game-playing AI to the playing strength of its opponent. It works by continuously tuning weights associated with rules found in agent-specific rule-bases, using the feedback received from online encounters, similarly to actor-critic approaches (Sutton and Barto, 2018).

In contrast, action preselection was designed as a preprocessing technique that aims to shrink and shape the search space, in line with a given strategy (expressed through parameters), for a subsequent planning phase. To that end, heuristics were used instead of rules as in dynamic scripting for two reasons. (1) Rules work in a narrow scope, and (2) will always impose a decision by returning a single action. Heuristics extend the scope of rules by supporting parameters, making a single heuristic act as a family of related rules. Moreover, a heuristic does not always impose a decision and can return multiple possible actions compatible with its goal. Such actions are forwarded to a planning algorithm to make a decision. Although online optimization is a possibility, in Chapter 5 we propose to optimize action preselection hyperparameters offline using an EA.

4.2 | Parametric Action Preselection

While playing an RTS game, a human player would generally interact with groups (or squads) of units more frequently than they would with individual units. We assume this behavior is required to cope with the continuously growing number of their units in the environment. For this reason, the user interfaces of most modern RTS games offer streamlined group management facilities. The player would then have to figure out a tactical objective for each group (harvest, reconnaissance, attack, defend, ..., etc.) and assign the correct task to each, in line with their intended global strategy. One way to model this behavior is through authoring rule-based scripts, which could implement a strategy by assigning actions to different groups of units.

A script, however, as intricate as it could get, cannot account for every possible type of opponent or map layout, and may get easily exploited by a robust search-based agent. Nevertheless, scripts remain as an interesting medium for encoding expert knowledge for use by more sophisticated RTS agents. We are particularly interested in the way rule-based scripts are constructed. If one closely inspects their inner-workings, it becomes obvious that such scripts are themselves a combination of smaller-scale, simple scripts, we denote as heuristics.

As an example, we inspect the Rush script. Rush is a well-known RTS script encoding a strategy that aims to overwhelm the opponent by constantly producing assault units and sending them immediately to attack the opponent's base. The WorkerRush variant can be broken down into three heuristics, the first one assigned to the Base **(Train)**, the second assigned to at least one Worker **(Harvest)**, and the third assigned to the remaining Workers **(Attack)**. The **Train** heuristic simply keeps training Workers as long as there are resources to cover the cost. The **Harvest** heuristic tasks its assignee Workers with harvesting and returning resources. The **Attack** heuristic will send the remaining Workers to attack the opponent's Base. The **Harvest** and **Assault** heuristics utilize a pathfinding algorithm to guide the units.

Viewing scripts as a combination of heuristics should help generate novel scripts because through this viewpoint, the attention now shifts to the question of how to design interesting heuristics and combine them to produce original scripts (or strategies). A script-generating system would help assist a search-based agent. We propose to conceive parameterized heuristics to facilitate the generation of novel heuristics, and consequently novel strategies, by simply defining the strategy's heuristics and their parameters. The possible advantages of a heuristic-based, parametric script-generation system include:

- **Explainable strategy authoring:** Selecting the right heuristics and their parameters can clearly describe the intended strategy.
- **Controllable output:** In contrast with regular scripts, parameterized heuristics can be configured to output more than one candidate action. A high-level search algorithm can decide which action to select.
- **Dynamic adaptation:** The strategy in use can be adapted in-game through heuristic re-parameterization or switching.
- **Parameter learning:** The presence of parameters allow for learning optimal parameter values for various situations. We propose to use an evolutionary algorithm for a similar task in Chapter 5.
- **Difficulty adjustment:** It is possible to define difficulty parameters for the set of heuristics and adjust them according to the player's level.

We propose to use a heuristics-based, script-generating system for an action preselection phase that precedes a NaïveMCTS search phase. This approach would signif-

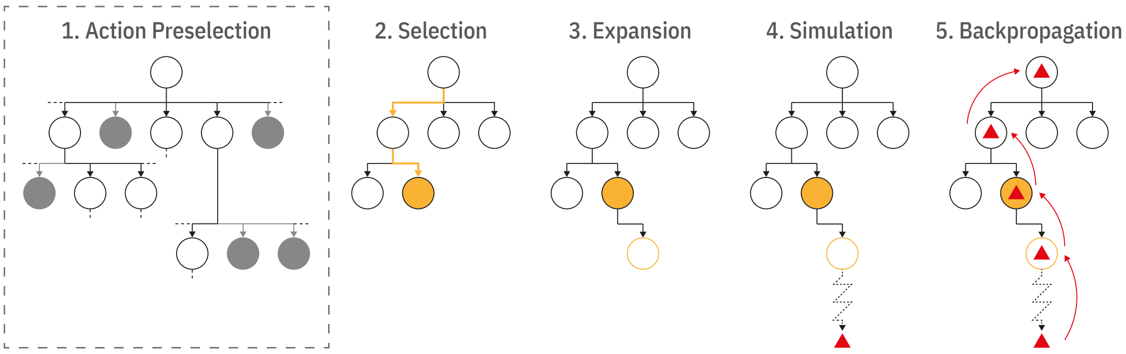


Figure 4.1: Action preselection as a prior step to the four phases of MCTS. Nodes represent states, and outbound edges represent actions. Actions and states discarded by preselection are shown in gray.

icantly lower the branching factor while framing the decision space according to the generated strategy. Moreover, this would also allow granular non-deterministic control thanks to the presence of multi-output heuristics.

Figure 4.1 shows the position of action preselection relative to MCTS phases. Action preselection is positioned as a pre-processing step that operates on the decision space before each MCTS iteration. Action preselection molds the decision space according to a strategy expressed using the combination of heuristics and their parameters. The result is a considerably smaller decision space, conserving a degree of granularity. NaïveMCTS, or possibly any search algorithm, operates as usual on the resulting decision space.

4.2.1 | Formal Definition

We define an arbitrary, parametric heuristic $h \in \mathcal{H}$, where $\mathcal{H}$ is the set of all RTS heuristics, as a function $h : S \times \mathcal{U} \times P(\mathcal{A}^u_s) \times \mathcal{R}_h \to P(\mathcal{A}^u_s)$ taking as input a game state $s \in S$, a unit $u \in \mathcal{U}$, all legal unit actions possible for u in s, represented by a set $\alpha \subseteq \mathcal{A}^u_s$ with $|\alpha| = l$, and a parameter vector $\mathbf{p} \in \mathcal{R}_h$. The output of h is a unit-action set $\alpha' \subseteq \mathcal{A}^u_s$ with $|\alpha'| = k$, under the constraints: $k \leq l$ and $\alpha' \subseteq \alpha$. If $k = l$, either the heuristic h works at the level of unit-action attributes or is an identity heuristic. $\mathcal{U}$ and $\mathcal{R}_h$ represent the sets of units and parameter vectors of heuristic h, respectively. $P(\mathcal{A}^u_s)$ denotes the power set of $\mathcal{A}^u_s$.

Non-parametric heuristics present in expert-authored scripts may implicitly encode constant parameter vectors. Parametric heuristics explicitly include those parameters as arguments to the heuristic. A parametric heuristic h is fully deterministic if $k = 1$, and no stochastic parameter controls its execution. In case $k > 1$, the heuristic is multi-output, and a search algorithm can select an optimal action from its output, α'. A

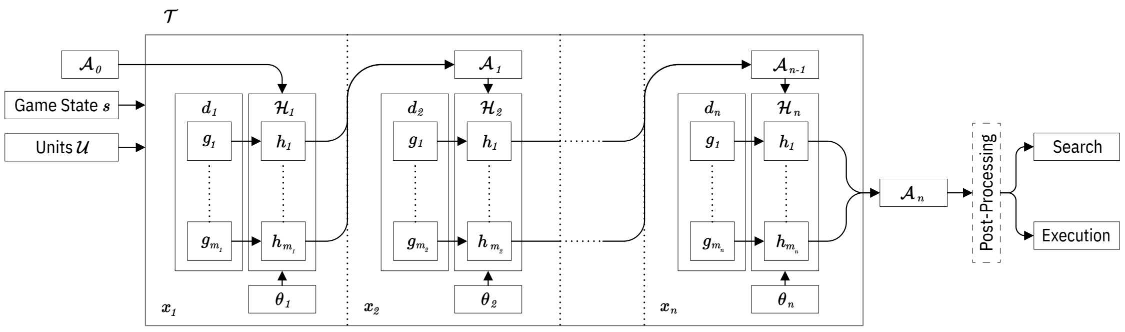


Figure 4.2: The action preselection process $\mathcal{T}$. Each phase refines the unit-actions set received from the previous phase, according to a strategy expressed by partitionings, heuristics, and parameters.

heuristic may employ any suitable algorithm, including pathfinding, to narrow down the number of actions in α' for the sake of reducing the overall branching factor. For instance, the *Harvest* heuristic will discard all unit-actions in favor of those that guide a *Worker* back and forth between a Resource Deposit and a Base. A parametric *Harvest* heuristic may expose parameters such as the maximum resources to harvest or the pathfinding algorithm to use.

A heuristic h, when associated with a group of units $g \in P(\mathcal{U})$, will be equally applied to each unit in g using the same parameter vector $\mathbf{p}$. Thus, a parametric group heuristic is denoted as $h[g, \mathbf{p}]$. We define $\mathcal{D}$ as the set of all possible unit partitionings, where a partitioning $d \in \mathcal{D}$ can be expressed as: $d = \{g_1, \cdots, g_m \mid g_i \in P(\mathcal{U}) \land u \in g_i \Rightarrow u \notin g_j, \forall j \neq i\}$. Determining a partitioning d can be done either manually or automatically. In the current implementation, we use a manual, first-come-first-served partitioning approach.

We define an action preselection process $\mathcal{T}(s, \mathcal{U}, \mathcal{A}_0, x_1, \cdots, x_n)$ as an n-phase algorithm operating on a given game state s, the set $\mathcal{U}$ of all units currently in s, and all their legal unit-actions $\mathcal{A}_0$. In any preselection phase, $x_i(\mathcal{A}_{i-1}, d_i, \mathcal{H}_i, \theta_i)$, each heuristic $h_j \in \mathcal{H}_i$ is applied to its assigned unit group $g_j \in d_i$, using the relevant parameters $\mathbf{p}_j \in \theta_i$, and the unit-actions resulting from the previous phase $\mathcal{A}_{i-1}$. The resulting unit-actions set, $\mathcal{A}_i$, is fed to the next preselection phase $x_{i+1}(\mathcal{A}_i, d_{i+1}, \mathcal{H}_{i+1}, \theta_{i+1})$. The output of the last phase, x_n, is also the final output of $\mathcal{T}$, and it represents the unit-actions that constitute the decision space framed by the strategy $\sigma_n = (d_x, \mathcal{H}_x, \theta_x)$, with $d_x = (d_1, \cdots, d_n)$, $\mathcal{H}_x = (\mathcal{H}_1, \cdots, \mathcal{H}_n)$, and $\theta_x = (\theta_1, \cdots, \theta_n)$. Figure 4.2 illustrates the preselection process $\mathcal{T}$. $\mathcal{A}_n$ may undergo an additional post-processing phase before it is used for search or execution.

Intuitively, an action preselection process is a successive refinement technique that sequentially manipulates the set of legal unit-actions of each unit, in the relevant game state, according to a global strategy expressed by σ_n. The resulting set of unit-actions represent a decision to execute, or the possible options admissible by σ_n. In the latter case, a search algorithm such as MCTS can be employed to find an optimal player-action in accordance with σ_n, in a much smaller and focused decision space. A regular expert-authored script can be seen as a one phase preselection process, σ_1, with deterministic, single-output heuristics.

As an example use-case, it is possible to define a hierarchical 2-phase strategy using the described action preselection process. In the first phase, units are split into two large groups, $d_1 = \{defense, offense\}$, that are assigned the heuristics $\mathcal{H}_1 = \{\textbf{defend}, \textbf{attack}\}$, under parameters θ_1. In the second phase, the units could get split into more specialized groups, such as $d_2 = \{baseDef, barracksDef, offense\}$, with associated heuristics $\mathcal{H}_2 = \{\textbf{defendBase}, \textbf{defendBarracks}, \textbf{attack}\}$ under parameters θ_2. The first phase ensures a common behavior for defense units, and the second phase builds on that to create specialized defense units. We will describe our proposed parametric action preselection implementation for RTS games next.

4.2.2 | Implementation: ParaMCTS

We propose to implement a versatile action preselection process, based on the most common heuristics employed by most RTS strategies. The general progression of any RTS match can be described as follows: A player starts by harvesting resources and building essential structures and at some point, the player must defend his base from the assaults of his opponent's forces and conduct assaults on his opponent's base. While confronting enemy units, the player must maximize the damage done, and minimize the damage taken, using astute micro-management. We attempt to model the heuristics in this general progression and allow for the emergence of interesting variants through heuristic parameterization.

The proposed implementation, ParaMCTS, uses NaïveMCTS to search in the decision space framed by a 2-phase action preselection process. The first phase partitions units into four **functional** groups, and the second phase puts units in two **situational** groups. Both phases are described next, in the context of µRTS.

4.2.2.1 | Phase 1: $x_1(\mathcal{A}_0, d_1, \mathcal{H}_1, \theta_1)$

This phase segregates units depending on their intended function, then assigns the relevant heuristic to each. d_1 keeps track of the number of units in each group, and assigns each unit to its relevant group, in a first-come-first-served fashion. The maximum number of units in each group is a partitioning parameter to provide. d_1 works in tandem with the Train heuristic to keep the number of units within the provided limits. Unit groups and heuristics are described bellow:

Functional Groups: d_1

- *Harvesters:* Worker units tasked with resource gathering and structure building.
- *Structures:* Barracks and Base. Responsible for training mobile units.
- *Offense:* Mobile units ready to assault opponent units anywhere.
- *Defense:* Mobile units assigned to defend the Base's perimeter.

Functional Heuristics: $\mathcal{H}_1$

- **Harvest:** Applies to the *Harvesters* group. Automates the resource harvesting process and provides Barracks building options whenever possible. Parameters include the building location selection mode (isolated or random), the maximum number of build options, and the pathfinding algorithm.
- **Train:** Applies to the *Structures* group. Trains units following the group composition in d_1. Parameters include d_1 group composition, the training side selection mode, and the maximum number of training options.
- **Attack:** Applies to the *Offense* group. Find and track opponent units for suppression. Parameters include the targeting mode (closest, minHP, ..., etc.), the maximum number of units to target, and the maximum number of escape routes to consider in a close encounter.
- **Defend:** Applies to the *Defense* group. Remain within a defense perimeter around the Base and attack incoming opponent units. Parameters include the geometry and size of the defense perimeter, the defense mode, and the maximum number of units to attack.

4.2.2.2 | Phase 2: $x_2(\mathcal{A}_1, d_2, \mathcal{H}_2, \theta_2)$

Using the output of the first phase, $\mathcal{A}_1$, the goal of this phase is to give a tactical advantage to the units in direct contact with the opposing forces, in an attempt to reproduce a micro-management behavior. d_2 works independently of d_1 and will group units according to their spatial situation with respect to enemy forces. The maximum number of units in each group is a partitioning parameter to provide.

Situational Groups: d_2

- *Front-Line:* A predefined number of mobile units in close-contact with opponent units. Selected randomly by scanning the fire-range of opponent or own units. Retrieves units in need of faster low-level reactivity.
- *Back:* All units not in the Front-Line group.

Situational Heuristics: $\mathcal{H}_2$

- **Front-Line Tactics:** Applies to the *Front-Line* group. Reduces the Wait unit-action duration to increase the units' reactivity while in combat.
- **Back Tactics:** Applies to the *Back* group. Keeps the default Wait unit-action duration.

All heuristics described thus far are simple, rule-based, and multi-output. Through parameter configuration (θ_1 and θ_2), the dimensionality and structure of the resulting decision space at the end of the last preselection phase, can vary substantially. Parameters must be selected in a way that generates the best strategy possible against a specific opponent in a specific map. NaïveMCTS is then tasked with searching within the decision space limited by the generated strategy for the best response possible. The multi-output heuristics ensure a degree of granularity within the decision space.

4.2.2.3 | Heuristic Switching

Switching heuristics on the fly is a way to adapt the agent's strategy according to environment changes. We propose to switch the heuristics of d_1's *Defense* group from **Defend** to **Attack** according to a conditional trigger. The switch is triggered whenever the player's army composition score surpasses the opponent's by a predefined margin we call *the overpower factor*. The army composition score is calculated by summing the resource costs of all mobile units of the relevant player. Intuitively, this switch occurs whenever the player has the chance to overwhelm its opponent. The switch is only triggered in the first preselection phase.

4.2.2.4 | Post-Processing

As a post-processing step, ParaMCTS applies IPA pruning to the final preselection output. This would help further diminish the branching factor and improve the reactivity of units. The percent of player-actions to prune constitutes an additional parameter to consider. For more details about IPA pruning, refer to Chapter 3.

4.2.2.5 | Parameters

In total, 46 parameters were defined between heuristics, partitionings, heuristic-switching, and post-processing. Appendix A describes the full list of ParaMCTS parameters. The reader can also refer to the source code repository[1] for more technical details. Before exploiting ParaMCTS, a manual parameterization phase must be carried out. The choice of parameters should account for the characteristics of the map, and the opponent's strategy.

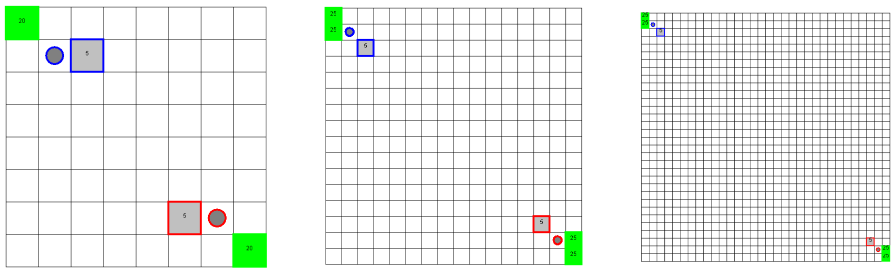
For the experiments described ahead, we configured ParaMCTS to adopt better, but non-exploitative, strategies against MixedBot (Mariño et al., 2018; Moraes et al., 2018), in multiple µRTS competition maps. In the next chapter we propose an automatic ParaMCTS configuration approach.

4.3 | Experiments & Results

The principal effect of action preselection manifests in the significant reduction of the branching factor, caused by discarding a large portion of unit-actions in favor of more meaningful ones. This alleviates the strain endured by NaïveMCTS in expansive combinatorial decision spaces, and should help improve game-tree exploration. The objective sought by our experiments is to determine which MCTS parameter could exploit the downsized decision space to achieve the highest performance gain. The NaïveMCTS component of ParaMCTS could profit from the reduced branching factor in two ways; by increasing its maximum search depth, or by increasing the duration of playouts (simulations). In both cases, one may expect more accurate action value estimates and higher quality decisions, which may translate to a better overall performance.

To find out which MCTS parameter, and which value, makes an effective use of the newly gained advantage, we test the performance of ParaMCTS using gradually increasing search depth and playout duration values. Specifically, we conduct our initial experiments using the values from the two sets: $depthVals = \{10, 15, 20, 30, 50\}$ and

[1] `https://github.com/Acemad/UMSBot`



(1) BasesWorkers8x8A (2) BasesWorkers16x16A (3) BasesWorkers32x32A

Figure 4.3: The maps used in the experiments. These maps feature a symmetric layout, and represent a gradual increase in complexity from the smallest map to the largest.

$durationVals = \{100, 150, 200, 300, 500\}$. Using the results of those experiments, we proceed next to a performance assessment experiment against a group of state of the art, and baseline agents.

ParaMCTS$(depth, duration)$ is defined as the ParaMCTS variant using $depth$ and $duration$ as the maximum search depth, and playout duration, respectively. Three maps representing increasingly larger branching factors were used, namely *basesWorkers* 8×8, 16×16, and 32×32, as shown in Figure 4.3. To guarantee a fair comparison, ParaMCTS was manually parameterized to not adopt exploitative strategies, even when there could be a possibility to exploit certain agents. In all experiments, the score of an agent is calculated likewise: $score = wins + draws/2$, then normalized between 0 and 100.

All experiments were executed on two computers with relatively similar hardware and software configurations, using the latest version of µRTS as of the 10th of July 2020. All agents were given a $100ms$ computation budget per cycle.

4.3.1 | Experiments 1 & 2: Search Depth and Playout Duration

To study the impact of deeper search and longer playouts on ParaMCTS, we ran two 120-iteration round-robin tournaments as part of the first experiment. The first tournament ran between each ParaMCTS$(depth, 100)$ variant, for each $depth \in depthVals$, with playout $duration$ fixed at 100 in each variant. And, the second tournament ran between each ParaMCTS$(10, duration)$ variant, for each $duration \in durationVals$, with $depth$ fixed at 10 in each variant. Fixed values, 10 and 100, for $depth$ and $duration$, respectively, are the default NaïveMCTS values. Results of this experiment are shown in Figure 4.4.

In the second experiment, we took all the possible $depth$ and $duration$ combinations from $depthVals \times durationVals$, and ran 100 matches (switching sides after 50 matches)

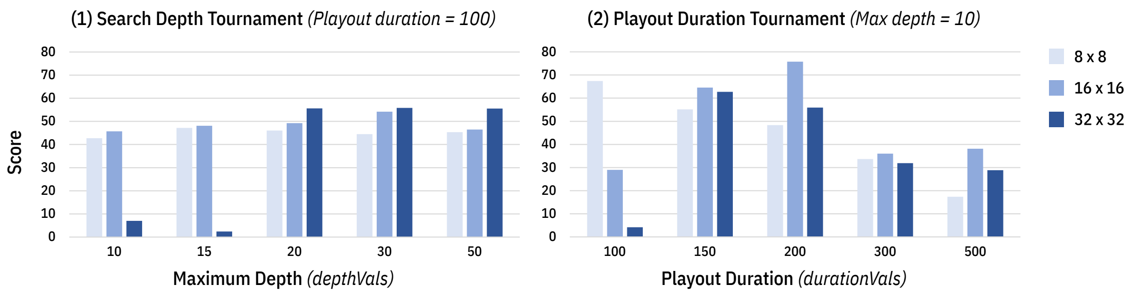


Figure 4.4: The results obtained by each ParaMCTS$(depth, duration)$ variant in both tournaments of the first experiment. The score represents the win rate of each variant against the other variants in the same tournament. (1) shows the results of the first tournament with a fixed playout duration (100) and a varying max depth, and (2) shows the results of the second tournament with a fixed max depth (10) and a varying playout duration.

between each resulting ParaMCTS$(depth, duration)$ agent, and MixedBot, a state-of-the-art agent combining various techniques (Barriga et al., 2019; Lelis, 2017; Mariño et al., 2018; Moraes et al., 2018). The results of this experiment in the three maps are presented in Table 4.1.

From the results of both experiments, we can see how the overall ParaMCTS performance seems to be particularly sensitive to the playout duration. In Figure 4.4-(2) performance vary significantly between playout durations. In the smallest maps short playouts work best, but in larger maps slightly longer playouts work well up to a certain threshold. As for search depth, it is clear that in the largest map a deeper search yields the most benefit, as seen in Figure 4.4-(1). As for the small and medium maps, deeper search holds fewer benefits.

Against MixedBot (Table 4.1), the best performance in all three map sizes is achieved when the playout duration equals 100 cycles, even if 150 cycles appears promising in the 16×16 map. If we consider the search depth, going down 20 levels in the tree is the optimal depth for 8×8 and 16×16 maps. In the 32×32 map, searching as deep as 50 levels produces the best performance. Evidently, a deeper search yields the highest performance gain than longer playouts. We believe this is true because deeper search could be responsible for more accurate player-action reward estimates, due to the increased number of visited nodes and playouts towards the depth of the game tree. On the other hand, longer playouts decrease the number of visited nodes and playouts, which may negatively impact performance. Larger maps benefit the most from deeper search because the map's dimensions contribute to the sparsity of rewards, and a deeper search can reach rewarding states more frequently.

Table 4.1: Results of the second experiment. Each square represents the score obtained by ParaMCTS($depth, duration$) against MixedBot. Cells with a score above 50 are gradually saturated in four levels: 50-59, 60-69, 70-79, and 80-89

Playout Duration

8 x 8

Max Depth	100	150	200	300	500	Avg
10	74.5	69.5	59	46	19	53.6
15	82	65	56	46	28.5	55.5
20	86	74	51	40	23	54.8
30	81	68	62	40	18	53.8
50	83.5	68	69	46	21	57.5
Avg	81.4	68.9	59.4	43.6	21.9	

16 x 16

Max Depth	100	150	200	300	500	Avg
10	3	74	63	20	20	36
15	1	80	71	10	28	38
20	88	77	59	15	25	52.8
30	71	73	66	9	21	48
50	68	74	62	4	29	47.4
Avg	46.2	75.6	64.2	11.6	24.6	

32 x 32

Max Depth	100	150	200	300	500	Avg
10	68.5	54.5	35.5	33	20.5	42.4
15	72.5	44	42	34	25.5	43.6
20	65.5	50.5	46	39.5	17.5	43.8
30	74	46	43	38.5	25	45.3
50	76.5	41	46.5	45.5	24	46.7
Avg	71.4	47.2	42.6	38.1	22.5	

Overall

Max Depth	100	150	200	300	500	Avg
10	48.7	66	52.5	33	19.8	44
15	51.8	63	56.3	30	27.3	45.7
20	79.8	67.2	52	31.5	21.8	50.5
30	75.3	62.3	57	29.2	21.3	49
50	76	61	59.2	31.8	24.7	50.5
Avg	66.3	63.9	55.4	31.1	23	

4.3.2 | Experiment 3: Comparison Against State-of-the-Art

To assess the overall performance of ParaMCTS, we perform a 100-iteration round-robin tournament between it and a number of best performing agents. The tournament includes three top ranking agents from 2019's µRTS competition, specifically, MixedBot, Izanagi (Mariño et al., 2018; Moraes et al., 2018), and Droplet (Yang and Ontañón, 2019b). All of these agents use some form of abstraction-assisted search. The participating ParaMCTS agent uses the optimal depth and duration values found in previous experiments for each respective map. The tournament also includes two baseline agents, NaïveMCTS, and NMCTS*, a NaïveMCTS variant using the same search depth and playout duration as ParaMCTS. Results of the tournament are presented in Table 4.2 and Figure 4.5.

In terms of overall performance, ParaMCTS outperformed all state-of-the-art agents by a sizable margin. ParaMCTS was able to achieve an 11.9 points margin over the 2nd best agent, Izanagi, and 19.1 points margin over the 4th best, MixedBot. Both agents

Table 4.2: Overall results of the third experiment's tournament. Row vs Column.

	ParaMCTS	Izanagi	Droplet	MixedBot	NMCTS*	NMCTS	Average
ParaMCTS		50.0	72.0	84.8	96.0	94.7	79.5
Izanagi	50.7		42.7	64.8	89.5	90.2	67.6
Droplet	30.2	53.3		45.0	89.7	90.2	61.7
MixedBot	22.8	32.8	53.5		96.8	96.0	60.4
NMCTS*	7.3	9.2	7.2	2.2		50.2	15.2
NMCTS	6.8	9.7	8.7	2.5	47.7		15.1
Average	23.6	31.0	36.8	39.9	83.9	84.3	

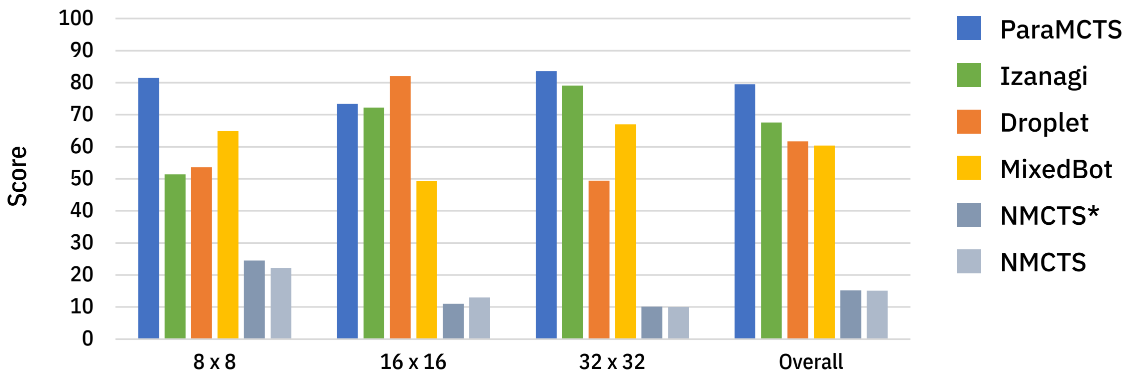


Figure 4.5: Third experiment tournament results by map. Overall score represents the average score.

make use of a combination of advanced techniques. This result is a direct evidence of the potency of our action preselection approach when coupled with NaïveMCTS. In individual maps, ParaMCTS outperformed the other agents in 8×8 and 32×32 maps, but it was overcome in the 16×16 map by Droplet. This can be explained by the adoption of an exploitative strategy by Droplet in the 16×16 map. Although ParaMCTS can be configured to adopt similar strategies, we chose not to do so in order to keep the comparison as fair as possible. NMCTS* did not offer any tangible performance gain over NaïveMCTS, which indicates that increasing the search's depth will not yield any performance gain if not paired with a significant decision-space reduction.

4.3.3 | UMSBot: ParaMCTS in the µRTS Competition

ParaMCTS took part in the 4th edition of the annual µRTS AI competition, held as a side event of IEEE® CoG 2020. University of Mustapha Stambouli Bot (UMSBot) was the alias of the participating ParaMCTS agent, which we registered for the Classic track, featuring full observability and determinism. The final results of the classic track tournament of the competition are reported in Table 4.3.UMSBot ranked fourth in the tournament, both in open maps, and in the combined open and hidden maps. Even if UMSBot could not rank among the top three agents, we believe that the achieved results are still remarkable. UMSBot managed to outperform all baseline agents, and two non-baselines, Rojo and GuidedRojoA3N, with the latter making use of the HyP method, A3N (Moraes and Lelis, 2018a). The average difference between UMSBot and the third highest-ranking agent, MentalSeal, is clearly non-significant. Against agents like UTS_Imass, Rojo, and POWorkerRush, UMSBot performed better than MentalSeal, which uses an enhanced version of GNS (Yang and Ontañón, 2019b). By significantly outperforming baseline NaïveMCTS, and using NaïveMCTS as the search algorithm, UMSBot has proven that NaïveMCTS, combined with the right decision and state space abstractions can produce a viable approach towards a strong RTS AI. This also indicates that our heuristics-based action preselection framework constitutes a promising path for search-based agents. It is worth noting that the winning agent, CoacAI, relies on fast, expert-authored and rule-based scripts.

4.4 | Summary

Throughout this chapter, we have introduced a different point of view related to the problem of domain-knowledge exploitation in RTS games. Instead of keeping with the trend of using full-blown monolithic scripts that encode expert-authored strategies, we switched our perspective towards the implicit components of scripts, the heuristics. We proposed to control heuristics in a granular fashion through parameters and use these parametric heuristics to formulate parameter-defined strategies. Next, we defined a system that frames the decision space of an RTS game, in successive phases, according to a strategy expressed by the combination of heuristics, unit partitioning schemes, and their respective parameters. A search algorithm could then exploit the downsized decision space to perform more efficient searches. We proposed a basic action preselection implementation, ParaMCTS, that relies on common domain knowledge for the design of its heuristics and parameters, then uses NaïveMCTS for search (Ouessai et al., 2020b).

The downsized decision space freed NaïveMCTS to conduct deeper searches, and

Table 4.3: Overall results of the Classic Track tournament of the 4th µRTS competition in all maps. Asterisk (*) marks baseline agents.

	CoacAI	UTS_Imass	MentalSeal	UMSBot	POLightRush*	Rojo	NaïveMCTS*	POWorkerRush*	GuidedRojoA3N	RandomBiased*	Average
CoacAI		65.8	95.0	97.5	100	99.2	100	94.2	100	100	94.6
UTS_Imass	34.2		97.1	93.3	98.3	94.6	98.8	89.2	99.2	99.6	89.4
MentalSeal	5	2.9		59.6	72.1	82.1	86.7	80.4	85.8	100	63.8
UMSBot	2.5	6.7	40.4		62.9	83.3	83.3	87.1	84.6	98.8	61.1
POLightRush*	0	1.7	27.9	37.1		60.4	77.1	77.1	62.1	94.6	48.7
Rojo	0.8	5.4	17.9	16.7	39.6		54.2	52.1	57.9	83.3	36.4
NaïveMCTS*	0	1.3	13.3	16.7	22.9	45.8		64.6	61.3	74.6	33.4
POWorkerRush*	5.8	10.8	19.6	12.9	22.9	47.9	35.4		48.8	86.3	32.3
GuidedRojoA3N	0	0.8	14.2	15.4	37.9	42.1	38.8	51.3		74.2	30.5
RandomBiased*	0	0.4	0	1.3	5.4	16.7	25.4	13.8	25.8		9.9
Average	5.4	10.6	36.2	38.9	51.3	63.6	66.6	67.7	69.5	90.1	

further enhance its playing strength. Experimentation results have shown a significant improvement over some state-of-the-art µRTS agents. ParaMCTS participated in the 2020 µRTS competition as UMSBot and managed to outperform all baseline agents, and two participating agents, making it one of the highest-performing MCTS-based agents in µRTS.

The proposed action preselection implementation, ParaMCTS, is a single possibility among many. Using action preselection as a basis to develop more sophisticated agents is a viable path ahead. Although proposed as a way to lower the RTS decision space dimensionality, we believe this technique could be easily adapted to any multi-unit real-time game. Real-time, in-game strategy adaptation is now a possibility for search-based agents, since heuristics and their parameters can be altered during a match. Automatically learning new heuristics, and finding better ways to combine them is also an interesting direction. As a byproduct, we believe this approach could be most useful in an industry setting. The self-contained nature of heuristics could help in facilitating the development of a scalable and maintainable AI system for commercial games, which could also rely on parameters for tasks such as difficulty adjustment or personalization.

In the next chapter, we investigate how to apply an Evolutionary Algorithm (EA) to optimize the parameters of ParaMCTS heuristics, and automatically generate stronger strategies that can outperform previously unbeatable agents.

5

Evolutionary Action Preselection

"However beautiful the strategy, you should occasionally look at the results"

— Winston Churchill

ParaMCTS requires a prior configuration phase relying on domain knowledge. The performance of ParaMCTS depends strongly on the configuration quality of the underlying action preselection process. Configuring an action preselection process occurs in two phases. In the first phase, we need to determine the heuristics that compose the process and their associated unit partitionings. In the second phase, we try to find a set of optimal values for the parameters of each heuristic and partitioning scheme. Up to this point, we used manual configuration in both phases. The design of ParaMCTS' preselection process was conceived manually by integrating heuristics and partitionings found in the general progression of an RTS game match. ParaMCTS parameters were tuned manually using MixedBot as a baseline in multiple µRTS competition maps. In both instances, we heavily relied on domain knowledge.

The excessive reliance on domain knowledge for configuring an action preselection agent can be disadvantageous for two reasons. First, the agent needs to be reconfigured for each new opponent and map combination to preserve its performance. Second, domain knowledge is not always readily available, and even if it is, it cannot guarantee an optimal configuration. Thus, action preselection applicability may suffer if domain knowledge is the sole source of configuration.

As a first step towards a fully automatic action preselection process, we propose to act on the second configuration phase using an Evolutionary Algorithm (EA) (Bäck, 1996) to automatically find optimal action preselection parameters under diverse circumstances. This could eliminate the domain-knowledge requirement in the specification of preselection parameters and replace it with an evolutionary optimization process for finding optimal parameters. Since each set of preselection parameters encodes a distinct strategy, our proposed approach effectively searches for an optimal strategy versus a specific opponent within a specific environment. In specific terms, we suggest the usage of a Genetic Algorithm (GA) to optimize the parameters of an action preselection agent when facing a strong state-of-the-art agent. We will closely examine the evolution of fitness across several maps, then validate the performance of the evolutionary agent in a final tournament against multiple agents within the same maps. The experiments' results in µRTS indicate that such approach can indeed discover strategies that surpass the manually-defined ones in a subset of tested environments (Ouessai et al., 2022).

5.1 | Related Works

The exploitation of EAs for optimizing game-playing agents is a common practice in game AI, and many techniques were proposed to deal with the resulting challenges (Lucas and Kendall, 2006). EAs are commonly used offline under supervised conditions, to optimize the parameters of an AI controller, before any online exploitation. This is also the case for controllers based on neural networks (Risi and Togelius, 2017). Nevertheless, EAs are also used as effective online AI controllers, as proven by Justesen et al. (2016)'s Online Evolution, and Perez et al. (2013)'s Rolling Horizon Evolution.

In the RTS games domain, the usage of EAs for offline optimization is more prevalent. Ponsen et al. (2005) evolved a population of rule-bases within the dynamic scripting framework (Spronck et al., 2006) for Wargus. Fernández-Ares et al. (2011) Used an EA to optimize the parameters of a rule-based bot playing the simple RTS game, Planet Wars. In a follow-up work Mora et al. (2012) studied the problem of noisy fitness faced by EAs when optimizing agent parameters. García-Sánchez et al. (2015) applied Genetic Programming (GP) to evolve complete strategies for StarCraft, and Fernández-Ares et al. (2017) later studied the impact of different fitness functions on the generated strategies. Louis and Liu (2018) optimized a micromanagement agent in a 3D RTS game, using a multi-objective evolutionary algorithm, and Gajurel et al. (2018) proposed a neuroevolution approach for RTS micromanagement. An evolutionary approach was proposed by Mariño et al. (2018) for finding an optimal action abstraction by evolving a population of agents, each producing a different abstraction layer using a different subset of scripts. Abstractions were evaluated by playing a subset-selection game against each other.

Fewer works were dedicated to online EAs in RTS games, compared to the offline counterpart. Wang et al. (2016) replaced PGS' hill climbing optimization with an EA for their Portfolio Online Evolution (POE) algorithm. In their experiments using StarCraft, POE successfully outperformed PGS. In the macro-management level, Justesen and Risi (2017) introduced an evolutionary real-time build order optimization approach known as Continual Online Evolutionary Planning (COEP).

We propose to use offline evolutionary optimization to find an optimal set of hyper-parameters for an action preselection process, under different situations. Our approach is similar to the aforementioned offline RTS agent optimization approaches, but differing in the underlying optimization target. Instead of scripts and rule-bases, we attempt to optimize the parameters of a set of heuristics, defined by the ParaMCTS action preselection implementation. We evaluate a parameter set against a high-performing agent in hopes of finding a stronger ParaMCTS variant. Our end goal is to eliminate manual knowledge-based parameter tuning in favor of an automatic approach.

5.1.1 | Relation with Hyper-Heuristics

A Hyper-Heuristic (Burke et al., 2003) is defined as any technique that uses (meta-) heuristics to select (meta-) heuristics for solving a given problem. In action preselection, a differently parameterized instance of the same heuristic can be seen as a standalone heuristic. Therefore, when we use an EA for parameter optimization, we are actually using a meta-heuristic to select a group of heuristics in an attempt to obtain an optimal RTS AI. Our approach, thus, is compatible with this broad definition of hyper-heuristics. The hyper-heuristics framework definition by Burke et al. (2003) eliminates the flow of domain knowledge between the hyper-heuristic and the low-level heuristics. In our proposition, we do supply the hyper-heuristic (the EA) with domain knowledge in the form of parameter constraints, specific to each low-level heuristic. As a result, our approach does not fully conform to the hyper-heuristics framework.

5.2 | Evolutionary Algorithms (EAs)

Evolutionary Algorithms (Bäck, 1996) define a category of optimization methods that attempt to model the evolutionary behavior of living organisms, with the aim to replicate their adaptation mechanisms. EAs can be seen as a form of directed random search, where solutions get randomly sampled and modified under the guiding influence of a fitness function. EAs share the same set of elementary evolutionary operators, known as selection, recombination, and mutation. The role of the selection operator is to ensure that high-quality solutions are most likely to survive and reproduce. Recombination forms new solutions by merging parts of multiple solutions in hopes of producing better quality solutions. Mutation introduces novelty through the random alteration of solutions' components. EAs are widely used for complex optimization tasks featuring a large search space, and a non-differentiable objective function. The four most popular Evolutionary Algorithms (EAs) are: Evolutionary Strategies (ESs), Evolutionary Programming (EP), Genetic Algorithms (GAs), and Genetic Programming (GP).

For the current proposition, we employ a Genetic Algorithm (GA) in an attempt to discover an optimal parameter set for our action preselection implementation. GAs were initially introduced by Holland (1984) as an abstract model of biological evolution, where a population of chromosomes, encoding potential solutions to a problem, evolves through the application of genetic operators to form a new, possibly more performant, population. GAs adopt the terminology and concepts of molecular genetics. Consequently, a candidate solution is represented by a collection of one or more chromosomes, and each chromosome is an array of genes, with each gene taking a single value from a set of values called alleles. The typical GA evolution process proceeds as follows:

1. Generate a population consisting of random chromosomes (solutions).
2. Evaluate the quality of the population's members using a fitness function f.
3. Use a selection operator to select a subgroup (parents) of chromosomes from the population. Apply crossover and mutation operators on the members of the subgroup, following a crossover probability p_c, and a mutation probability p_m.
4. Use a reinsertion strategy to introduce the new offspring population back into the original population.
5. Repeat the process using the new population, starting from the evaluation phase (2), until a stopping criterion is reached.

The birth of a new population marks the start of a new generation. A frequently used stopping criterion is the determination of whether a maximum number of generations is attained. Throughout the evolution process, the mean fitness of the population of potential solutions is expected to gradually improve. Ultimately, a possibly optimal solution should emerge under the constraints of the fitness landscape. The encoding of candidate solutions plays a major role in defining the dimensionality of the search space, and the quality of the solutions. Usually, the best individual of the last generation is the expected to represent an optimal solution.

5.3 | Evolving Action Preselection Parameters

To successfully exploit the advantages of ParaMCTS, it is required to configure its parameters for each map and opponent combination. This is unrealistic if we intend to conceive the most general agent possible. In a competition environment, where there is a fixed number of maps and opponents, it is possible to use domain knowledge to manually tune heuristics against strong opponents in every competition map. This is how we proceeded in the 2020 µRTS competition, and our entry UMSBot (A ParaMCTS instance tuned manually against MixedBot, a strong asymmetric-abstraction based agent), ranked fourth in the overall Classic-track ranking.

Besides the possibly non-generalizable performance, manual parameter configuration accentuates the domain-knowledge dependency, already heavily relied-on in the design of ParaMCTS heuristics. Furthermore, domain-knowledge is not something that is constantly available, especially for new maps and opponents, and even if it is, the resulting parameter combination does not guarantee an optimal performance. This situation calls for an automatic parameter configuration approach that may discover an optimal set of parameter values for any map/opponent combination. The ParaMCTS parameter search space can be exceedingly large. If we assume that each parameter can take only 2 different values, this would amount to $2^{46} \approx 10^{13}$ possible combination. An approximative optimization algorithm should be able to find an optimal parameter combination. Therefore, we propose to use a GA for this task.

We focus principally on whether the parameters found by the GA can outperform the manually-tuned parameters. To utilize a GA, we initially need to find a way to encode any potential set of action preselection parameters as a solution (genotype). Next, we must determine how to accurately evaluate the quality of those solutions.

5.3.1 | Encoding

Our objective is to discover a set of parameters that could form the best performing ParaMCTS agent possible. This translates to a Combinatorial Optimization Problem (COP) where each potential solution corresponds to a vector p structured as a combination of values; one for each parameter in $\theta_1 \cup \theta_2 \cup \theta_0 \cup \theta_N$. We define θ_0 as the set containing heuristic-switching parameters, post-processing parameters, and the partitioning parameters for d_1 and d_2. θ_N is the set of NaïveMCTS ϵ-greedy parameters.

To solve this COP using a GA, the first step is to establish an appropriate solution encoding for p. We use a real (or value) encoding, which means that the direct representation of the parameters is used, and each parameter is encoded as a single gene. Because

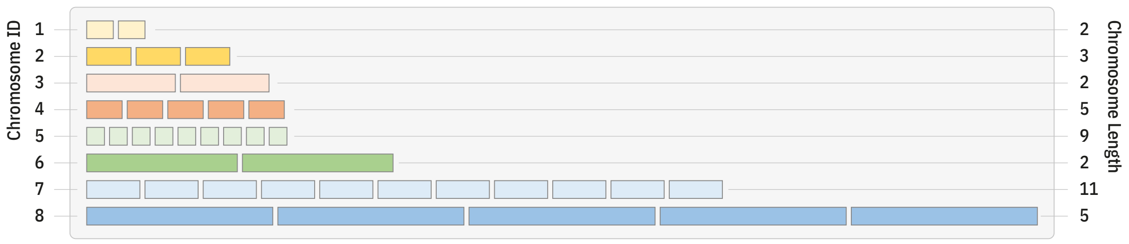


Figure 5.1: The encoding of a genotype. A genotype groups eight (8) chromosomes, each having at least two genes. The size of a gene correlates with the range of values it can encode.

ParaMCTS' parameters do not share a common range of values, multiple chromosomes should be used to group similarly-constrained genes together. The final genotype of a solution p comprises eight (8) chromosomes as shown in Figure 5.1. In this figure, each chromosome is identified by an ID and a length (the number of genes). The longer the gene the wider the range of values it admits.

In an effort to hasten the evolution process, we kept the 36 most impactful parameters out of the 46 ParaMCTS parameters, in addition to the three parameters in θ_N, for a total of 39 parameters/genes. The parameters/genes mapping, and the specifics of the used and unused parameters can be found in Appendix A.

5.3.2 | Fitness Function

A precise evaluation of the quality of an individual is an important consideration to address while applying a GA. Fitness evaluation is responsible for directing search towards optimal regions in the search space, and a fitness function of poor accuracy could misguide search and reduce it to a plain random process. To evaluate the quality of an RTS agent, usually a large number of matches is executed versus one or more target agents, while ensuring to switch the starting positions after completing 50% of the matches, to guarantee fairness. The average score obtained by the agent indicates its quality. Because RTS matches are non-deterministic, this type of evaluation provides a rather accurate estimate of the agent's performance as the number of matches grows. However, the accuracy of the evaluation would decline as the number of matches decreases.

In the context of GAs, evaluating each individual by playing a large number of matches would result in an extremely time-consuming evolution, which could be impractical, even if the accuracy is high. Seeking a trade-off between the speed of evolution and the accuracy of evaluation is the main concern to deal with when conceiving a fitness function for this kind of problem. We use the fitness function $f_n(c) = v$ to evaluate individuals. f_n takes a phenotype c (A ParaMCTS agent instance based on the genotype of c) and returns a value $v \in [0, 1]$. n corresponds to the number of matches to play against a target agent on a target map. n must be an even number to ensure an equal number of matches in both starting sides for the two agents. The outcome of a match is either $+1$ for a win, -1 for a loss, or 0 for a draw. v is the average of the outcomes of all n matches, normalized in $[0, 1]$. This is a common case of noisy fitness that arises while attempting to optimize game-playing agents for games of non-deterministic outcomes. In the context of RTS games, Mora et al. (2012) and Fernández-Ares et al. (2017) conducted a thorough study of this particular phenomenon.

5.3.3 | Genetic Operators

We used a generational GA, as implemented by the Jenetics (Franz Wilhelmstötter, 2020) library used in our experiments. It is a standard implementation that begins by selecting parents and survivors, and applying recombination and mutation operators to the parents. The resulting offspring individuals substitute their parents and combine with the survivors to form the new population. The process iterates until a stopping criterion is reached. Because of the evaluation noise existing in the proposed fitness function, it is important to choose a selection operator with the lowest possible sensitivity to noise. Therefore, we used the tournament selector for parents and survivors selection due to its decent performance in noisy fitness landscapes (Miller and Goldberg, 1995). Tournament selection uses a number of competitors s.

With a computationally costly fitness function, it would be beneficial to use recombination operators capable of stimulating a higher convergence rate. Uniform crossover seems to be best suited for this type of problem because of its ability to better explore the parameter space. Uniform crossover is applied to every pair of selected parents with a probability p_c. A regular uniform mutation operator is applied to each resulting offspring individual with a probability p_m. The GA begins with a population consisting of m random individual and selects a proportion r and $1 - r$ from m to form the parents and survivors set, respectively. The stopping criterion is triggered when the number of generations reach a predefined limit g.

5.4 | Experiments & Results

The objective of our experiments is to find out whether a GA would succeed in automatically discovering a set of ParaMCTS parameters that can deliver a superior perfor-

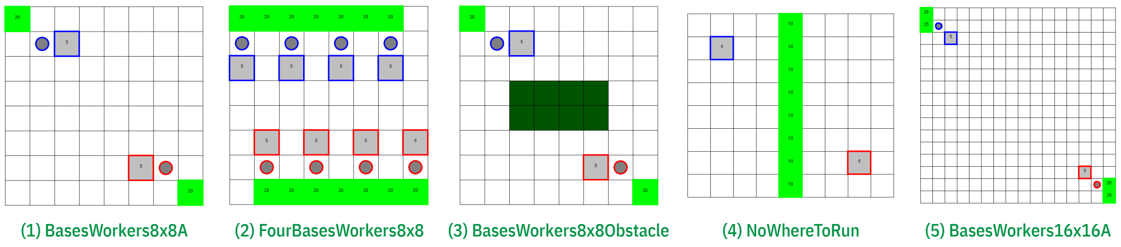


Figure 5.2: The maps used in the experiments. Maps (2), (3), and (4) could pose an interesting challenge to parameter tuning. The dark green squares in map (3) represent walls.

mance than the manually-tuned ones. In the first experiment we use a GA, as described in the previous section, to evolve a population of ParaMCTS parameters, for the sake of finding a better-parameterized ParaMCTS agent with respect to a target agent and a set of maps. We call the agent resulting from this evolution stage, EvoPMCTS. In the second experiment, we attempt to validate the performance of EvoPMCTS by running a round-robin tournament against multiple agents in the same set of maps.

A set of five distinct maps were used, two from the previous chapters' experiments, and an additional three representing interesting situations that necessitate non-trivial strategies. The maps are shown in Figure 5.2, and all of them are consistently present in the µRTS competition, except for map (3). EvoPMCTS will take advantage of the results of Chapter 4 experiments and use the best search depth and playout duration parameters found. Because all maps are smaller than 16×16, a maximum search depth of 20, and a playout duration of 100 cycles should be adequate.

5.4.1 | Experiment 1: Evolving Preselection Parameters

We propose an evolutionary optimization approach to explore the parameter space for more interesting parameter combinations. We used the Jenetics library (Franz Wilhelmstötter, 2020) configured using the solution encoding, and the fitness function defined in the previous section. The fitness function f_n was adapted to evaluate an individual by running $n = 10$ matches against a single strong opponent in a given map from those in Figure 5.2. The value of n was selected empirically after initial tests, because it was found to offer a satisfactory evaluation accuracy in a reasonable time, compared to lower/higher values. We chose CoacAI[1] as the strong opponent that plays the role of the optimization target along the maps. CoacAI was the winner of the 2020 µRTS competition Classic track, in which ParaMCTS participated. It is a fast, expert-designed

[1]https://github.com/coac/coac-ai-microrts

Table 5.1: The parameters of the GA used for EvoPMCTS parameter optimization.

Population Size (m)	Generations Count (g)	Crossover Rate (p_c)	Mutation Rate (p_m)	Selection: Parents Fraction (r)	Selection: Tournament Competitors (s)
20	100 or 450	0.7	0.2	0.6	4

rule-based agent incorporating strong strategies for various µRTS maps. CoacAI makes for the best choice as an evaluation opponent, because of its speed and strength. However, it remains to be seen whether a strong agent could help in the emergence of a stronger EvoPMCTS agent. A single evaluation match is limited to 3000 cycles for maps (1) to (4), and 4000 cycles for map (5). If no winner is determined before those limits are reached the match ends in a draw.

We performed a single GA run for each of the five maps in Figure 5.2 using the configurations detailed above and the GA parameters presented in Table 5.1. The choice of the GA parameters was made with a single consideration in mind. That is, **the evolution process should be able to discover a solution, surpassing baseline quality, within a reasonable computation time.** After a series of exhaustive tests, we settled on the parameters in Table 5.1, which appeared to satisfy our consideration. A small population size ($m = 20$) reduces the number of evaluations needed, keeping evolution time in check, while high mutation ($p_m = 0.2$) and crossover ($p_c = 0.7$) rates ensure a decent exploration capacity. In the majority of maps, 100 generations were enough to witness individuals overcome baseline fitness. Otherwise, we permitted evolution to reach the 450th generations.

5.4.1.1 | The Computational Cost

The resulting computational cost of this experiment depends principally on the cost of the evaluation function f_n, which revolves around the time complexity of the evaluation matches. The cost of one match is computed by multiplying the match's length (in game cycles) by the sum of the computational budget spent by the two agents (CoacAI and EvoPMCTS) in each game cycle. We suppose the cost of µRTS game-mechanics processing is negligible. Each agent is given $100ms$ as a computational budget, and CoacAI being a rule-based agent, only uses roughly $10ms$ on average (according to our observation), totaling $110ms$ per cycle for both agents. The estimated computational cost of the evolutionary process with respect to a single map can be computed as such: $g \times m \times n \times c \times 110ms$. Assuming we know the average match length, c, for the relevant map. For map (1), we estimated c to be 595 cycles, which gives a computational cost of

15.12 days of computation time. In the larger map (5), c was estimated to be 977 cycles, giving a computational cost of 24.87 days. For maps (2), (3), and (4), the estimates are equivalent to those of map (1). We used eight processor threads to perform evaluations in parallel, and the computation cost diminished to 1.89 days for map (1) and to 3.10 days for map (5).

In total, 260000 evaluation match were carried out across the five maps. Individuals were reevaluated after each iteration in an attempt to offset the effect of noisy fitness, which substantially increased the total matches count. The resulting fitness evolution plots for each map are shown in Figure 5.3.

5.4.1.2 | Analysis

First, we determine the baseline fitness as the median value 0.5, over which EvoPMCTS starts outperforming CoacAI. The optimal fitness value is the maximum value the fitness function can take, namely 1. In all evolution plots, the effect of noise in the fitness function is clearly exhibited through the fluctuations in mean, min, and max fitness values. More importantly, in maps (1) to (4), the max fitness begins under the baseline value, then successfully fluctuates above it. In map (5), though, the max fitness starts at the optimal value and remains there during the remaining generations. If EvoPMCTS is represented by a max fitness individual, we can see that it is able to attain optimal fitness in maps (1), (2), and (5). As for maps (3) and (4), EvoPMCTS could overtake the baseline fitness with a high margin. This does not yet validate the performance of EvoPMCTS versus CoacAI due to the small number of matches executed for fitness evaluation ($n = 10$). Nevertheless, it does confirm that the GA-based optimization is able to find EvoPMCTS agents that surpass the baseline fitness and even attain optimal fitness under rational conditions.

The difficulty degree, v, is defined as the number of generations the max fitness needs to overtake the baseline. Using v, we can assess and compare the difficulty of the maps from the point-of-view of EvoPMCTS (and ParaMCTS, by extension). Therefore, we can point out three map difficulty levels from the evolution plots. Map (5) represents a *trivial* difficulty level with $v = 1$, whereas maps (1), (2), and (3) are of *regular* difficulty, each with a v equaling 6, 9, and 4, respectively. The map having the highest difficulty degree is (4), with $v = 74$, depicting a *challenging* difficulty level for EvoPMCTS. Note that this map-specific difficulty degree depends principally on the design of the inherent action preselection process, including heuristics and their parameters.

In the *trivial* difficulty map (5), max fitness attains the optimal value instantly after

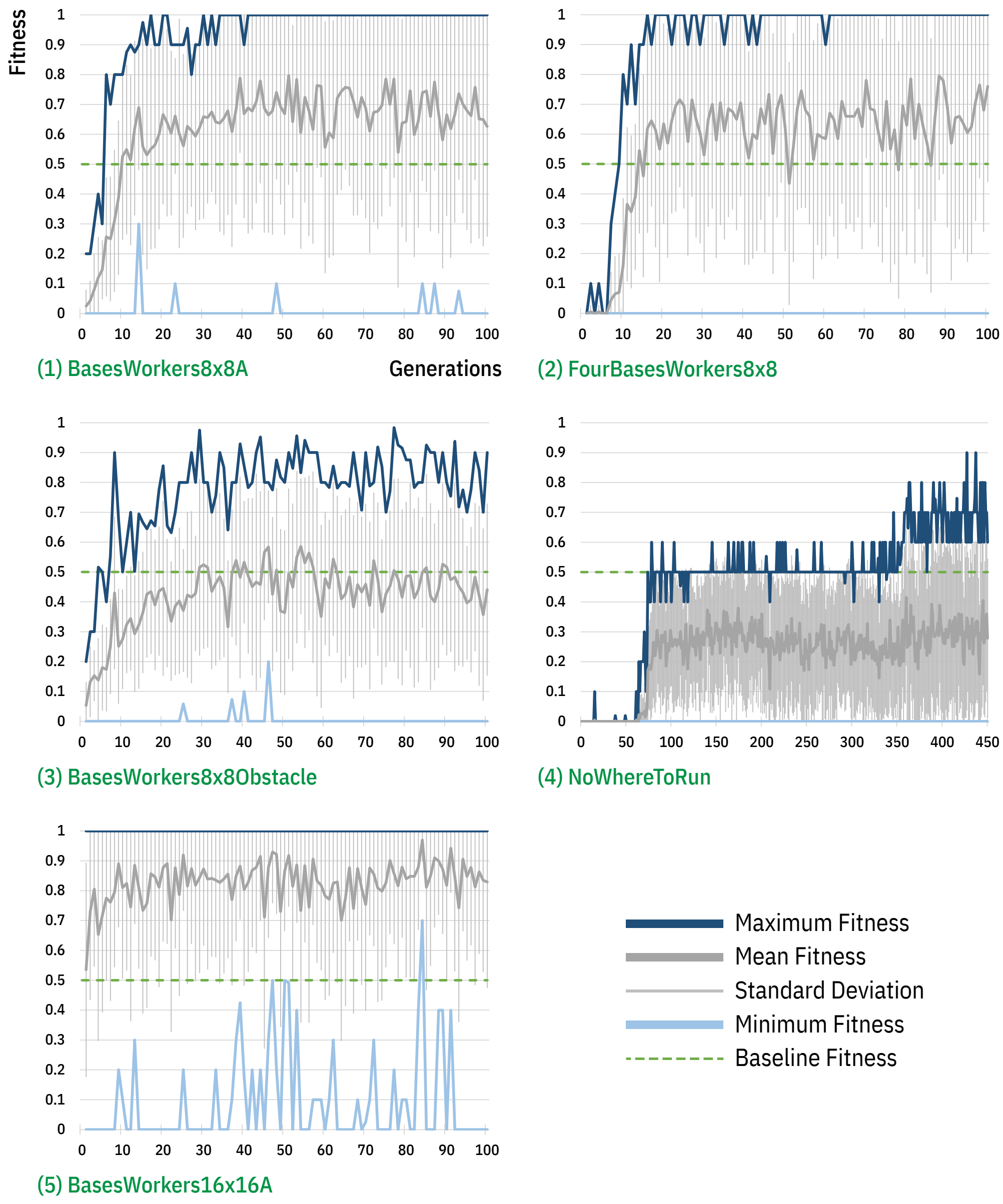


Figure 5.3: The evolution of fitness throughout the generations in each map. Each plot concerns one of the maps and shows the evolution of the maximum fitness, mean fitness, standard deviation, and minimum fitness of the population. The fitness represents the performance of the evolving EvoPMCTS agent against CoacAI.

the first generation. One possible explanation may suggest that an optimal individual already existed in the initial population, which implies that random sampling may be enough to acquire an optimal EvoPMCTS agent for map (5). Nevertheless, if we examine the mean fitness, we can notice a small upwards trend, and likewise, the minimum fitness seems to fluctuate towards higher fitness. This indicates that the initial, seemingly optimal, individual may get replaced by another one with a more stable fitness in subsequent generations, meaning that optimization may still be useful in *trivial* maps such as (5).

In the *challenging* map (4), and as opposed to map (5), the max fitness could not attain the baseline until the 74th generation. In hopes of reaching optimal fitness, we permitted the evolution to continue up to the 450th generation, however the optimal fitness kept out of reach. After the 74th generation, max fitness remained stuck between suboptimal values, not far from the baseline, for over 270 generations. Afterwards, fitness progressed towards higher suboptimal values and fluctuated around there until the end of evolution. It is unclear whether more generations would help EvoPMCTS achieve optimal fitness in this map. The explanation we can provide concerning this behavior is that there may be some noisy parameters in the EvoPMCTS parameter space that could be misleading search and stalling the evolution process. It would be interesting to see whether a better-designed parameter space could mitigate this issue.

In the remaining *regular* difficulty maps, (1), (2), and (3), max fitness rapidly overtakes the baseline and eventually attains optimal fitness, aside from map (3), where it stays fluctuating near the optimal fitness. In maps (1) and (2), max fitness stabilizes in the end, analogously to map (5). Similarly, we suppose that this stabilization phase helps in improving the fitness stability of the resulting EvoPMCTS agent. As for map (3), which features a closely similar layout as map (4), interestingly, EvoPMCTS was not able to attain optimal fitness and seems to exhibit a similar, but much less severe, struggle as in map (4). Thus, we can conclude that the map's layout can be crucial in determining how easily EvoPMCTS could attain optimal fitness. In maps (1), (2), and (5), where a clear open area separates the opponents' bases, EvoPMCTS succeeds in attaining optimal fitness. However, in maps (3) and (4), where physical obstacles stand between the opposing sides, the optimal fitness is more challenging to attain. The existence of obstacles apparently exposes a weakness in the design of EvoPMCTS (and eventually ParaMCTS) possibly related to the absence of heuristics that account for obstacles.

In the following experiment, EvoPMCTS' performance, based on the highest performing individuals from each evolutionary process, will be examined against state-of-the-art agents to verify the potential of the evolved strategies.

5.4.2 | Experiment 2: Performance Validation

In order to determine whether the parameters obtained through the evolutionary optimization processes could produce high-performing EvoPMCTS agents, we ran a round-robin tournament in each of the formerly used maps using the following agents:

- **EvoPMCTS:** The ParaMCTS agent configured using the best parameters discovered by the evolutionary process, for each map.
- **UMSBot:** The manually-configured ParaMCTS agent, tuned versus MixedBot. Attained the fourth position in classic track of the 2020 µRTS competition.
- **CoacAI:** The winner of the classic track in the 2020 µRTS competition. A fast, expert-designed, rule-based agent used as the evaluation opponent in EvoPMCTS optimization phase.
- **MixedBot:** A script-assisted search-based agent combining three distinct agents (Barriga et al., 2019; Lelis, 2017; Mariño et al., 2018; Moraes et al., 2018). Ranked second in the 2019 µRTS competition (Classic track).
- **NaïveMCTS:** A baseline LLP algorithm, based on Combinatorial Multi-Armed Bandits (CMABs) (Ontañón, 2017).

Each tournament ran for 100 iteration, and in each iteration every agent plays two matches, one in each side, against the remaining agents for a total of 20 matches per iteration, and 2000 match per round-robin tournament. In total, 10000 matches were played. The normalized scores obtained for each agent in each map are reported in Table 5.2.

The first observation we can make from the data in Table 5.2 is the outstanding performance of EvoPMCTS versus CoacAI. In all maps, except (4), EvoPMCTS was able to outperform CoacAI, often by a large margin. In map (4), EvoPMCTS reached a nearly equal performance to CoacAI, probably because the challenges encountered in the optimization phase for this specific map translated into a somewhat weaker performance. By contrast, the score of UMSBot (Manually-configured ParaMCTS) versus CoacAI is null in maps (2) and (3), and significantly low in maps (1), (4), and (5). This result validates two points. First, the evolutionary optimization process can generate EvoPMCTS agents with a scalable performance by executing only 10 matches per fitness evaluation. Second, the parameters obtained through evolutionary optimization could indeed outperform the manually-tuned parameters, for this specific use-case. Overall, the strate-

Table 5.2: Results of the final tournament in the five maps, read as row vs column. The overall results indicate the average score of the agents across the five maps. Framed rows highlight the scores of EvoPMCTS.

	CoacAI	UMSBot	EvoPMCTS	MixedBot	NaïveMCTS	Average
CoacAI		97.5	35	100	100	83.13
UMSBot	2.5		90.75	67.25	69.5	57.50
EvoPMCTS	65	9.25		46	61.5	45.44
MixedBot	0	32.75	54		88.75	43.88
NaïveMCTS	0	30.5	38.5	11.25		20.06

(1) BasesWorkers8x8A

	EvoPMCTS	CoacAI	UMSBot	NaïveMCTS	MixedBot	Average
EvoPMCTS		71.5	97.5	100	99.5	92.13
CoacAI	28.5		100	100	100	82.13
UMSBot	2.5	0		58.5	71.5	33.13
NaïveMCTS	0	0	41.5		45.25	21.69
MixedBot	0.5	0	28.5	54.75		20.94

(2) FourBasesWorkers8x8

	CoacAI	EvoPMCTS	MixedBot	UMSBot	NaïveMCTS	Average
CoacAI		44	98	100	96.5	84.63
EvoPMCTS	56		82.5	33.5	81.25	63.31
MixedBot	2	17.5		58	76	38.38
UMSBot	0	66.5	42		30.5	34.75
NaïveMCTS	3.5	18.75	24	69.5		28.94

(3) BasesWorkers8x8Obstacle

	CoacAI	UMSBot	MixedBot	EvoPMCTS	NaïveMCTS	Average
CoacAI		58.75	66.75	50.75	100	69.06
UMSBot	41.25		57	81.75	84.5	66.13
MixedBot	33.25	43		81	86.5	60.94
EvoPMCTS	49.25	18.25	19		95.75	45.56
NaïveMCTS	0	15.5	13.5	4.25		8.31

(4) NoWhereToRun

	CoacAI	EvoPMCTS	UMSBot	MixedBot	NaïveMCTS	Average
CoacAI		41.5	87	91	100	79.88
EvoPMCTS	58.5		65.5	99.75	0	55.94
UMSBot	13	34.5		72	100	54.88
MixedBot	9	0.25	28		100	34.31
NaïveMCTS	0	100	0	0		25.00

(5) BasesWorkers16x16A

	CoacAI	EvoPMCTS	UMSBot	MixedBot	NaïveMCTS	Average
CoacAI		39.95	88.65	91.15	99.30	79.76
EvoPMCTS	60.05		44.80	69.35	67.70	60.48
UMSBot	11.35	55.20		61.95	68.60	49.28
MixedBot	8.85	30.65	38.05		81.20	39.69
NaïveMCTS	0.70	32.30	31.40	18.80		20.80

Overall

gies generated by the GA appear to be much more effective than the manually-defined ones, at least against CoacAI.

Furthermore, EvoPMCTS also managed to significantly outperform MixedBot in maps (2), (3), and (5), with a higher score than the one obtained by UMSBot, which was manually-tuned against MixedBot. Even though our goal was not to optimize for the most general agent possible, this results shows that a degree of generalization is still achievable if a strong agent (CoacAI) is used as an evaluation opponent. Against NaïveMCTS, EvoPMCTS comfortably obtains higher scores than UMSBot, except in map (1) and (5) where UMSBot does better. In map (5) in particular, EvoPMCTS is completely dominated by NaïveMCTS despite the high performance versus the three remaining agents, which could signify that the strategy learned for this map overfits the strategies implemented by the script-based agents, leaving at least an opening that might be exploited by a robust search agent. Interestingly, UMSBot can outperform EvoPMCTS in maps (1), (3), and (4) which indicates that a generalizable performance is still unreachable. The last two observations are understandable, knowing that the optimization phase relied on a single target opponent for fitness evaluation, effectively searching for the strategies that produce the best response against that particular agent (CoacAI).

Concerning map-wise average performance, CoacAI outranks all other agents in maps (1), (3), (4), and (5), as expected. The only case where CoacAI gets outranked occurs in map (2), where EvoPMCTS ranks first with a 10 points margin above CoacAI. In this particular map, EvoPMCTS appears to have discovered an optimal strategy that works effectively against all tested agents by exploiting the strategy adopted by CoacAI. That is another case of possible generalization. In maps (3) and (5) EvoPMCTS ranks above UMSBot in further evidence of the superiority of the GA-based parameter tuning, with respect to manual tuning. Nonetheless, in maps (1) and (4), EvoPMCTS is only superior to baseline NaïveMCTS, which could be understandable for map (4) since its related optimization process never reached the optimal fitness. But for map (1), EvoPMCTS reached optimal fitness, and despite that, it acts as in map (4) and fails to expand the performance gain against UMSBot or MixedBot. This might be another instance of generalization failure caused by the use of a single agent as an evaluation target in the optimization phase, possibly producing overfitted strategies.

In the case of overall results, EvoPMCTS outranked UMSBot, which represents a clear positive answer to our initial question of whether automatically-tuned parameters could outperform the manually-tuned ones. Therefore, we can conclude that a GA-based optimization process is able to discover the right set of EvoPMCTS (eventually, ParaMCTS) parameters that could express a strategy capable of overcoming the evaluation opponent in a more substantial manner than manually-tuned parameters. Although such approach was not conceived to obtain highly generalizable strategies, in some instances the generated strategies could maintain a satisfying performance across diverse opponents. From these experiments, we can also affirm that the action preselection implementation, ParaMCTS, (the base technique used in UMSBot and EvoPMCTS) is capable of composing a wide array of different strategies, as witnessed by the very different performances of UMSBot and EvoPMCTS. ParaMCTS is hardly the best possible implementation. A better-designed action preselection approach could produce more interesting strategies and generate more promising evolutionary agents.

5.5 | Summary

In this chapter, we proposed to further enhance the action preselection process described in Chapter 4, by eliminating the problematic manual parameter tuning phase, in favor of an automatic evolutionary optimization process. A GA was employed to find an optimal parameter combination against a single strong agent, CoacAI, in multiple maps. The resulting EvoPMCTS agent was able to reach optimal fitness in open-area maps but could only surpass the baseline and maintain a suboptimal fitness in maps split by obstacles. In a final tournament, EvoPMCTS exhibited a robust performance against CoacAI and the manually-parameterized ParaMCTS agent, UMSBot. Despite not being optimized for generalized performance, EvoPMCTS was still able to maintain a high performance versus different opponents.

The findings in this chapter could present several interesting outcomes. First, the overall obtained results constitute a positive assessment for the aptitude of the heuristic-based action preselection approach in multi-action real-time games. Therefore, it could be considered as a valid approach to consider in such domains. Second, the design of heuristics and their parameters holds a direct influence on the performance of the agent, thus, ParaMCTS could get easily outperformed by a better-designed action preselection implementation with more sophisticated heuristics. Third, the layout of the environment (map) could require specific heuristics to attain acceptable performance. Fourth, an optimization algorithm (e.g. GA) could indeed discover interesting parameter combinations that may outrank expert-determined parameters, and that answers the question we posed earlier in this chapter.

EvoPMCTS is still constrained by the action preselection process design of ParaMCTS, meaning it would not be able to generate strategies that require some behavior outside the scope of ParaMCTS heuristics. Action preselection is still limited by expert knowledge when it comes to the design and combination of heuristics. It would be interesting to also explore ways to automate the process of designing and combining heuristics prior to parameter configuration.

6

Conclusion

"The peak efficiency of knowledge and strategy is to make conflict unnecessary."

— Sun Tzu

The principal focus of this thesis gravitates around the fundamental online adversarial planning challenge faced by an RTS game-playing AI. We proposed a number of approaches to enhance the compatibility and the performance of the MCTS algorithm in the RTS games domain. The inspiration that prompted us to follow this line of research was the super-human performance achieved by MCTS in the high-dimensionality decision/state space of Go (Silver et al., 2016). We attempted to enhance a successful MCTS variant, NaïveMCTS, born from the same inspiration, by applying a move pruning approach, introducing an integrated action/state abstraction framework, and applying an EA to learn strategies within the parameter space of the said framework.

Move pruning is a common game-tree search technique popular in the domain of board games AI. Our move pruning implementation for MCTS focused on a type of decisions (IPAs) that rarely offer any advantage, but negatively participates in the inflation of the decision space. We proposed four move pruning algorithms targeting this particular type of actions. The performance gain achieved in NaïveMCTS and UCT was significant, especially in larger maps with a high branching factor. In some instances, UCT even approached the performance of NaïveMCTS. To our knowledge, this was the first application of move pruning in the context of game-tree search in RTS games. Moreover, the approach is simple and easy to implement in any game with a combinatorial decision space and idle-type actions.

State and action abstraction is another promising direction that may prove crucial towards better MCTS applicability in RTS games. In that regard, we proposed to decompose expert-scripts into a collection of small-scale scripts we called heuristics. Heuristics guide a group of units towards the realization of a predetermined objective. We have shown that heuristics can be enhanced by adopting parameters and allowing multiple possible actions as an output. We introduced an action preselection process that executes before MCTS' selection phase and applies a heuristics-driven, successive action refinement routine, to filter and shape the decision space before MCTS planning. The ParaMCTS implementation, with the help of the integrated NaïveMCTS, was able to exploit the downsized decision space to search deeper. ParaMCTS exhibited a remarkable performance against state-of-the-art agents, both in our experiments and in the µRTS competition, as UMSBot.

ParaMCTS faces a domain knowledge bottleneck that could limit its potential and applicability. In an effort to reduce its reliance on domain knowledge, we made use of an EA to automatically find a set of ParaMCTS parameters, in hopes of outperforming a strong target agent in a set of given maps. The EA had to be tuned correctly to account for the noisy fitness related to the non-deterministic outcome of RTS matches. The resulting agent, EvoPMCTS, managed to surpass the performance of the manually-tuned ParaMCTS, and even CoacAI, 2020 µRTS competition winner, in some instances. This approach has proven to be effective in finding better parameters, and yet, it suffers from an overfitting problem, limiting generalization. Nevertheless, the results of our experiments confirm the viability of evolutionary action preselection as an alternative to previous abstraction approaches relying on coarse scripts. They also validate our hypothesis stating that an EA can find better parameters than the manually-tuned ones.

We believe the research works presented in this thesis contribute to the RTS AI, and the general AI, state of the art. By introducing move pruning to the RTS games domain, we demonstrated how a simple heuristic can significantly increase the performance of a decision space sampling approach, without any side effect or negative computational impact. Through action preselection, we have shown that small-scale parameterized scripts (heuristics) can be used to form a more effective abstraction layer that also keeps a degree of granularity for the subsequent planning phase. Manual configuration of the action preselection process can be replaced by an automatic evolutionary optimization phase that yields possibly better strategies, as proven empirically in our experiments.

Although the proposed approaches were conceived primarily for the RTS games domain, we strongly believe that there are valid applications of the said approaches in other domains as well. Any domain featuring a group of autonomous agents that must work together towards the completion of a strategic or tactical objective, within an adversarial environment, can benefit from the approaches we have proposed. Examples of such domains may include multi-drone military operations, robots-assisted warehousing and logistics, autonomous vehicles fleets, and satellite coordination. Agents can also belong to the same entity, for instance, the actuators/sensors of a robot or a self-driving vehicle. Obviously, an adaptation phase must precede any application attempt, especially if MCTS is envisaged for planning, which requires fast simulations. A simulator can be easily replaced by a surrogate model, however special care must be taken to guarantee the accuracy of the model. MCTS as a holistic approach has demonstrated a solid performance in highly complex domains which makes it an attractive alternative.

6.1 | Future Research Perspectives

There is plenty of room for improvements, and a lot of questions to address and investigate regarding the proposed approaches. We propose the following non-exhaustive list of possibly interesting directions to pursue for future research:

- IPA pruning proved the existence of detrimental actions in the RTS decision space, we believe there are more detrimental actions to uncover. A thorough analysis of the low-level decision space should help in finding and pruning more of those actions.
- The proposed action preselection process uses a simple unit grouping method. Using an intelligent clustering method may produce better results.
- The number of RTS heuristics is not out of reach. Implementing all possible RTS heuristics and building a heuristics' library could greatly help produce more performant agents. One may study how to combine heuristics from the library to form good preselection agents.
- It is possible to learn expert heuristics from professional match replays of commercial RTS games. What machine learning approach can be used to do so?
- The effect of using other search algorithms in the decision space resulting after action preselection is not yet known. A comparative study would be useful.
- Dynamic adaptation of RTS agents is one of the most important challenges to AI. Action preselection includes the necessary basics to implement dynamic strategy adaptation. However, which technique can be used to intelligently adapt the parameters of action preselection in real-time in order to respond to the opponent's strategy.

- How can we adapt the proposed approaches for a partially observable setting? How to adapt a heuristic for partial observability, and what heuristics are needed?
- The proposed approaches were tested and validated in µRTS because of the absence of a reliable and fast forward model (a.k.a. simulator) in commercial games. Implementing an approximate forward model, or eliminating the need for one, would provide an opportunity to examine the effect of our approaches in commercial games, and other domains, such as control and robotics.
- Implement and test the proposed approaches as part of StarCraft II using a surrogate forward-model.
- Currently, RTS AI suffers from the lack of an accurate state evaluation function. What can possibly be done to discover a more accurate RTS state evaluation function?

A

ParaMCTS Parameters

This appendix is intended as a reference, shortly describing specific information concerning the parameters of each component in the proposed parametric action preselection implementation, ParaMCTS. The parameters are distributed across four Tables, and each table focuses on a group of related components. Table A.1 and A.2 detail Phase 1 and Phase 2 heuristics' parameters, respectively. Table A.3 describes the parameters of heuristic-switching and post-processing, and Table A.4 concerns NaïveMCTS parameters. In each row of each table we provide (1) a parameter ID, (2) the name of the parameter, (3) a short description, (4) the range of possible values, (5) a default value, and (6) the ID of the encoding chromosome, based on Figure 5.1.

A total of 49 parameters are listed, including those of NaïveMCTS. 39 parameters were used for the evolutionary optimization phase, and the ten unused parameters are displayed in a faded tone within the tables. The principal reason for excluding these parameters was to decrease the size of the search space. We targeted these ten parameters due to their potential to inflate the search space, thanks to the wide range of values they could take, all while offering little or no strategic/tactical advantages. The ten parameters were omitted from the encoding of genotypes. However, in the phenotypes, they were assigned fixed values, equaling the default values indicated. Each of the 39 remaining parameters control a strategic or tactical aspect of clear impact.

Discarded parameters 6, 7, 20, 21, and 22 in Table A.1 concern the isolated build/train method, which searches for a build/train cell surrounded by the least amount of occupied cells, in hopes of minimizing the chances of deadlocks. We have selected fixed values that worked well in preventing deadlocks. The rectangular defense perimeter was kept as the only usable type of defense perimeters after deactivating circular perimeters through parameter 33 (Table A.1). For heuristic-switching (Table A.3), three types of triggers are available. We only used the score-trigger to switch defense units to offense, and fixed the value of assault units by parameter 42 for the calculation of the army composition score. Defense-time and unit-count triggers were deactivated by parameters 43, 44, and 45. For the unused parameters, n depicts an undefined upper-bound value, where applicable.

As part of the genotype, every parameter takes a discrete integer value. Two types of parameters exist, cardinal and ordinal. Cardinal parameters encode a quantity, like a unit count or a distance between two entities (E.g. parameters 8-16 and 28-29). On the other hand, ordinal parameters encode a choice as an integer, such as pathfinding algorithms, or the defense or offense modes (E.g. parameters 5, 25, 27, 31, and 32). The values that may be taken by ordinal parameters are restricted by the available options. For cardinal parameters, we have selected an upper-bound for each in a way that minimizes combinatorial explosion and keeps strategic and tactical diversity. For some parameters, special value, -1, is used to bypass the upper-bound if necessary. Cardinal unit-count parameters 1-2, 8-16, and 35 are additionally used as partitioning parameters for d_1 and d_2.

The parameters encoded in chromosome 8 (40 and 46-49) represent a discretization of the continuous parameters of score-trigger heuristic switching, post-processing, and NaïveMCTS. In the phenotype, these parameters get multiplied by 0.05 to obtain a real value in the [0,1] range.

Table A.1: Properties of Phase 1 heuristics parameters (θ_1)

Harvest Heuristic					
ID	Parameter	Description	Values	Default Value	Chromosome
1	maxBases	*The max. number of bases allowed*	0 ... 2	1	1
2	maxBarracks	*The max. number of barracks allowed*	0 ... 2	1	1
3	buildLocation	*Build-cell location selection method*	0: Random, 1: Isolated	1	5
4	maxBuildActionsChosen	*The max. number of Build actions allowed*	1 ... 4	2	4
5	harvestPathFinder	*The pathfinding algorithm for harvester units*	0: AStar, 1: Flood fill	0	5
6	isolatedBuildScanRadius	*The radius of the zone to scan, centered at a possible build-cell location*	0 ... n	1	
7	isolatedBuildMaxOccupiedCells	*The max. number of occupied cells tolerated in the scanned zone*	0 ... n	1	

Train Heuristic

ID	Parameter	Description	Values	Default Value	Chromosome
8	maxHarvesters	*The max. number of harvesting Worker units*	-1 … 4 (-1: Unlimited)	1	7
9	maxOffenseWorkers	*The max. number of offense Worker units*	-1 … 4 (-1: Unlimited)	-1	7
10	maxOffenseLights	*The max. number of Light offense units*	-1 … 4 (-1: Unlimited)	0	7
11	maxOffenseRanged	*The max. number of Ranged offense units*	-1 … 4 (-1: Unlimited)	0	7
12	maxOffenseHeavies	*The max. number of Heavy offense units*	-1 … 4 (-1: Unlimited)	0	7
13	maxDefenseWorkers	*The max. number of defense Worker units*	-1 … 4 (-1: Unlimited)	0	7
14	maxDefenseLights	*The max. number of Light defense units*	-1 … 4 (-1: Unlimited)	3	7
15	maxDefenseRanged	*The max. number of Ranged defense units*	-1 … 4 (-1: Unlimited)	-1	7
16	maxDefenseHeavies	*The max. number of Heavy defense units*	-1 … 4 (-1: Unlimited)	2	7
17	trainSide	*Structure-side selection method for new units*	0: Random, 1: Isolated	1	5
18	maxTrainActionsChosen	*The max. number of Train actions allowed*	1 … 4	2	4
19	priority	*Per-stance unit production priority*	0: Defense, 1: Offense	0	5
20	isolatedTrainScanWidth	*Width of the zone to scan, centered at a cell beside a structure*	0 … n	1	
21	isolatedTrainScanDepth	*Depth of the zone to scan, starting from a cell beside a structure*	0 … n	2	
22	isolatedTrainMaxOccupiedCells	*The max. number of occupied cells tolerated in the scanned zone*	0 … n	2	

Attack Heuristic

ID	Parameter	Description	Values	Default Value	Chromosome
23	maxTargetsOnOffense	*The max. number of opponent units to target*	1 … 4	2	4
24	maxEscapes	*The number of move actions allowed while attacking*	0 … 4	1	2
25	offenseTargetMode	*Opponent units targeting method*	0: Closest to Self, 1: Closest to Base, 2: Min HP, 3: Max HP, 4: Random	0	2
26	fixedTarget	*Constantly targeted opponent unit(s)*	1: None, 2: Base, 3: Barracks, 4: All Structures	3	4
27	offensePathFinder	*The pathfinding algorithm for offense units*	0: AStar, 1: Flood fill	0	5

Defend Heuristic

ID	Parameter	Description	Values	Default Value	Chromosome
28	horizontalDistanceFromBase	*Rectangular Defense Perimeter: The horizontal distance from base*	0 … MapWidth (0: Disabled, if V-distance is also null)	3	6
29	verticalDistanceFromBase	*Rectangular Defense Perimeter: The vertical distance from base*	0 … MapHeight (0: Disabled, if H-distance is also null)	3	6
30	maxTargetsOnDefense	*The max. number of opponent units to target*	1 … 4	2	4
31	defenseMode	*The defense objective*	0: Defend Base, 1: Defend Self	0	5
32	defensePathFinder	*The pathfinding algorithm for defense units*	0: AStar, 1: Flood fill	1	5
33	radiusFromBase	*Circular Defense Perimeter: The radius of the perimeter centered at the base*	0 … *max* (MapWidth, MapHeight) (0: Disabled)	0	

Table A.2: Properties of Phase 2 heuristics parameters (θ_2)

Front-Line Tactics Heuristic					
ID	Parameter	Description	Values	Default Value	Chromosome
34	frontLineSelectionMode	*Front-line units selection method (uses ATK range)*	0: Unit Range, 1: OPP Units Range	0	5
35	maxFrontLineUnits	*The max. number of front-line units to consider*	-1 ... 4 (-1: All)	3	7
36	frontLineTacticalDistance	*A distance added to the attack range to admit units that may soon confront an opponent unit*	0 ... 4	1	2
37	frontLineWaitDuration	*The duration of the Wait action for front-line units*	1 ... 10	3	3
38	frontLinePathFinder	*The pathfinding algorithm for front-line units*	0: AStar, 1: Flood fill	1	5

Back Tactics Heuristic					
ID	Parameter	Description	Values	Default Value	Chromosome
39	defaultWaitDuration	*The duration of the Wait action for back units*	1 ... 10	10	3

Table A.3: Properties of heuristic-switching and post-processing parameters.

Heuristic Switching					
ID	Parameter	Description	Values	Default Value	Chromosome
40	scoreTriggerOverpowerFactor	*The relative difference in army composition scores of the two players required to trigger a stance switch*	0 ... 20	5	8
41	switchingUnitCount	*The number of Defense units allowed to switch stance*	-1 ... 4 (-1: All)	-1	7
42	scoreTriggerAssaultUnitValue	*The value of assault units w.r.t. worker units in army composition score calculation*	1 ... n	4	
43	timeTriggerDefensePeriod	*The max. number of cycles defense units remain in a defensive stance before a stance switch*	-1 ... n Cycles (-1: Disabled)	-1	
44	timeTriggerSwitchDelay	*The number of cycles in which units can switch stance following a defense-time trigger*	0 ... n Cycles (0: Disabled)	0	
45	unitCountTriggerThreshold	*The number of defense units required to trigger a stance switch*	-1 ... n (-1: Disabled)	-1	

Post-Processing					
ID	Parameter	Description	Values	Default Value	Chromosome
46	ipaAllowProb	*The probability of allowing Inactive Player Actions*	0 ... 20	0	8

Table A.4: Properties of NaïveMCTS parameters.

NaïveMCTS					
ID	Parameter	Description	Values	Default Value	Chromosome
47	epsilon0	*The value of epsilon-0*	0 ... 20	8	8
48	epsilonG	*The value of epsilon-global (global MAB)*	0 ... 20	0	8
49	epsilonL	*The value of epsilon-local (local MAB)*	0 ... 20	6	8

References

Adams, E. *Fundamentals of Game Design 3rd*. New Riders, Berkeley, CA, 3rd edition, 2014a. ISBN 978-0-321-92967-9.

Adams, E. *Fundamentals of Strategy Game Design*. New Riders, 2014b. ISBN 978-0-13-381267-1 978-0-13-381201-5.

Andersen, P.-A., Goodwin, M., and Granmo, O.-C. 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 149–156, August 2018. doi: 10.1109/CIG.2018.8490409.

Antuori, V. and Richoux, F. Constrained optimization under uncertainty for decision-making problems: Application to Real-Time Strategy games. *arXiv:1901.00942 [cs]*, January 2019.

Arneson, B., Hayward, R. B., and Henderson, P. Monte Carlo Tree Search in Hex. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):251–258, December 2010. ISSN 1943-068X, 1943-0698. doi: 10.1109/TCIAIG.2010.2067212.

Auer, P., Cesa-Bianchi, N., and Fischer, P. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, 47 (2):235–256, May 2002. ISSN 1573-0565. doi: 10.1023/A:1013689704352.

Bäck, T. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press, January 1996. ISBN 978-0-19-535670-0.

Balla, R.-K. and Fern, A. UCT for Tactical Assault Planning in Real-Time Strategy Games. In *International Joint Conference on Artificial Intelligence*, pages 40–45, 2009.

Barriga, N. A., Stanescu, M., and Buro, M. Building Placement Optimization in Real-Time Strategy Games. In *Artificial Intelligence in Adversarial Real-Time Games: Papers from the AIIDE Workshop*, page 6, 2014.

Barriga, N. A., Stanescu, M., and Buro, M. Combining Strategic Learning with Tactical Search in Real-Time Strategy Games. In *AIIDE'17*, pages 9–15, Snowbird Ski Resort, Utah, September 2017.

Barriga, N. A., Stanescu, M., Besoain, F., and Buro, M. Improving RTS Game AI by Supervised Policy Learning, Tactical Search, and Deep Reinforcement Learning. *IEEE Computational Intelligence Magazine*, 14(3):8–18, August 2019. ISSN 1556-603X, 1556-6048. doi: 10.1109/MCI.2019.2919363.

Barriga, N. A., Stanescu, M., and Buro, M. Puppet Search: Enhancing Scripted Behavior by Look-Ahead Search with Applications to Real-Time Strategy Games. In *AIIDE'15*, pages 9–15, Santa Cruz, California, September 2015.

Barton, M. The History of Computer Role-Playing Games Part 2: The Golden Age (1985-1993). *Gamasutra*, February 2007.

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–49, March 2012. ISSN 1943-068X, 1943-0698. doi: 10.1109/TCIAIG.2012.2186810.

Burke, E., Kendall, G., Newall, J., Hart, E., Ross, P., and Schulenburg, S. Hyper-Heuristics: An Emerging Direction in Modern Search Technology. In Glover, F. and Kochenberger, G. A., editors, *Handbook of Metaheuristics*, International Series in Operations Research & Management Science, pages 457–474. Springer US, Boston, MA, 2003. ISBN 978-0-306-48056-0.

Buro, M. ORTS: A Hack-Free RTS Game Environment. In Schaeffer, J., Müller, M., and Björnsson, Y., editors, *Computers and Games*, Lecture Notes in Computer Science, pages 280–291, Berlin, Heidelberg, 2003a. Springer. ISBN 978-3-540-40031-8.

Buro, M. Real-time strategy gaines: A new AI research challenge. In *IJCAI 2003*, 2003b.

Buro, M. Call for AI research in RTS games. In *Proceedings of the AAAI Workshop on AI in Games*, pages 139–141. AAAI Press, 2004.

Campbell, M., Hoane, A. J., and Hsu, F.-h. Deep Blue. *Artificial Intelligence*, 134(1):57–83, January 2002. ISSN 0004-3702. doi: 10.1016/S0004-3702(01)00129-1.

Churchill, D. and Buro, M. Build Order Optimization in StarCraft. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 7(1), October 2011. ISSN 2334-0924.

Churchill, D. and Buro, M. Portfolio greedy search and simulation for large-scale combat in starcraft. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, Niagara Falls, ON, Canada, August 2013. IEEE. ISBN 978-1-4673-5311-3 978-1-4673-5308-3. doi: 10.1109/CIG.2013.6633643.

Churchill, D., Saffidine, A., and Buro, M. Fast Heuristic Search for RTS Game Combat Scenarios. In *Proceedings, The Eighth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pages 112–117, 2012.

Churchill, D., Preuss, M., Richoux, F., Synnaeve, G., Uriarte, A., Ontañnón, S., and Čertický, M. StarCraft Bots and Competitions. In Lee, N., editor, *Encyclopedia of Computer Graphics and Games*. Springer International Publishing, Cham, 2016. ISBN 978-3-319-08234-9. doi: 10.1007/978-3-319-08234-9_18-1.

Churchill, D., Buro, M., and Kelly, R. Robust Continuous Build-Order Optimization in StarCraft. In *IEEEE Conference on Games 2019*, page 8, 2019.

Clarke-Willson, S. The Origin of Realtime Strategy Game on PC. *The Rise and Fall of Virgin Interactive. Above the Garage Productions*, August 1998.

Dan Adams. IGN: The State of the RTS. *IGN*, April 2006.

Duguépéroux, J., Mazyad, A., Teytaud, F., and Dehos, J. Pruning Playouts in Monte-Carlo Tree Search for the Game of Havannah. In *Computers and Games*, volume 10068, pages 47–57. Springer International Publishing, Cham, 2016. ISBN 978-3-319-50934-1 978-3-319-50935-8. doi: 10.1007/978-3-319-50935-8_5.

Fencott, P. C., editor. *Game Invaders: The Theory and Understanding of Computer Games*. Wiley, Hoboken, N.J, 2012. ISBN 978-0-470-59718-7.

Fernández-Ares, A., Mora, A. M., Merelo, J. J., García-Sánchez, P., and Fernandes, C. Optimizing player behavior in a real-time strategy game using evolutionary algorithms. In *2011 IEEE Congress of Evolutionary Computation (CEC)*, pages 2017–2024, June 2011. doi: 10.1109/CEC.2011.5949863.

Fernández-Ares, A., Mora, A., García-Sánchez, P., Castillo, P., and Merelo, J. Analysing the influence of the fitness function on genetically programmed bots for a real-time strategy game. *Entertainment Computing*, 18:15–29, January 2017. ISSN 18759521. doi: 10.1016/j.entcom.2016.08.001.

Franz Wilhelmstötter. Jenetics: Java Genetic Algorithms Library. http://jenetics.io, 2020.

Gabriel, I., Negru, V., and Zaharie, D. Neuroevolution based multi-agent system for micromanagement in real-time strategy games. In *Proceedings of the Fifth Balkan Conference in Informatics*, BCI '12, pages 32–39, New York, NY, USA, September 2012. Association for Computing Machinery. ISBN 978-1-4503-1240-0. doi: 10.1145/2371316.2371324.

Gajurel, A., Louis, S. J., Méndez, D. J., and Liu, S. Neuroevolution for RTS Micro. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8, August 2018. doi: 10.1109/CIG.2018.8490457.

García-Sánchez, P., Tonda, A., Mora, A. M., Squillero, G., and Merelo, J. Towards automatic StarCraft strategy generation using genetic programming. In *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 284–291, Tainan, Taiwan, August 2015. IEEE. ISBN 978-1-4799-8622-4. doi: 10.1109/CIG.2015.7317940.

Geib, C. W. and Goldman, R. P. Recognizing Plans with Loops Represented in a Lexicalized Grammar. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI'11*, pages 958–963, 2011.

Geib, C. W. and Kantharaju, P. Learning Combinatory Categorial Grammars for Plan Recognition. In *The Thirty-Second AAAI Conference on Artificial Intelligence (AAAI-18)*, pages 3007–3014, 2018.

Geryk, B. A History of Real-Time Strategy Games. *GameSpot*, April 2011.

He, S., Wang, Y., Xie, F., Meng, J., Chen, H., Luo, S., Liu, Z., and Zhu, Q. Game Player Strategy Pattern Recognition and How UCT Algorithms Apply Pre-knowledge of Player's Strategy to Improve Opponent AI. In *2008 International Conference on Computational Intelligence for Modelling Control & Automation*, pages 1177–1181, Vienna, Austria, 2008. IEEE. ISBN 978-0-7695-3514-2. doi: 10.1109/CIMCA.2008.82.

Heinz, E. A. Adaptive Null-Move Pruning. *ICGA Journal*, 22(3):123–132, January 1999. ISSN 1389-6911. doi: 10.3233/ICG-1999-22302.

Hoki, K. and Muramatsu, M. Efficiency of three forward-pruning techniques in shogi: Futility pruning, null-move pruning, and Late Move Reduction (LMR). *Entertainment Computing*, 3(3):51–57, August 2012. ISSN 1875-9521. doi: 10.1016/j.entcom.2011.11.003.

Holland, J. H. Genetic Algorithms and Adaptation. In Selfridge, O. G., Rissland, E. L., and Arbib, M. A., editors, *Adaptive Control of Ill-Defined Systems*, NATO Conference Series, pages 317–333. Springer US, Boston, MA, 1984. ISBN 978-1-4684-8941-5.

Huang, J. and Yang, W. A multi-size convolution neural network for RTS games winner prediction. In Wang, Y., editor, *MATEC Web of Conferences*, volume 232, page 01054, November 2018. doi: 10.1051/matecconf/201823201054.

Huang, J., Liu, Z., Lu, B., and Xiao, F. Pruning in UCT Algorithm. In *2010 International Conference on Technologies and Applications of Artificial Intelligence*, pages 177–181, Hsinchu City, TBD, Taiwan, November 2010. IEEE. ISBN 978-1-4244-8668-7. doi: 10.1109/TAAI.2010.38.

Justesen, N. and Risi, S. Continual online evolutionary planning for in-game build order adaptation in StarCraft. In *Proceedings of the Genetic and Evolutionary Computation Conference on - GECCO '17*, pages 187–194, Berlin, Germany, 2017. ACM Press. ISBN 978-1-4503-4920-8. doi: 10.1145/3071178.3071210.

Justesen, N., Tillman, B., Togelius, J., and Risi, S. Script- and cluster-based UCT for StarCraft. In *2014 IEEE Conference on Computational Intelligence and Games*, Dortmund, Germany, August 2014. IEEE. ISBN 978-1-4799-3547-5. doi: 10.1109/CIG.2014.6932900.

Justesen, N., Mahlmann, T., and Togelius, J. Online Evolution for Multi-action Adversarial Games. In Squillero, G. and Burelli, P., editors, *Applications of Evolutionary Computation*, Lecture Notes in Computer Science, pages 590–603. Springer International Publishing, 2016. ISBN 978-3-319-31204-0.

Kantharaju, P., Ontanon, S., and Geib, C. W. μCCG, a CCG-based Game-Playing Agent for μRTS. In *IEEE Conference on Computational Intelligence and Games (CIG)*, 2018.

Kantharaju, P., Ontañón, S., and Geib, C. W. Extracting CCGs for Plan Recognition in RTS Games. In *Proceedings of the Workshop on Knowledge Extraction in Games 2019*, 2019a.

Kantharaju, P., Ontañón, S., and Geib, C. W. Scaling up CCG-Based Plan Recognition via Monte-Carlo Tree Search. In *IEEE Conference on Games 2019*, August 2019b.

Kirby, N. *Introduction to Game AI.* Course Technology/Cengage Learning, Boston, 2011. ISBN 978-1-59863-998-8.

Kocsis, L. and Szepesvári, C. Bandit Based Monte-Carlo Planning. In Fürnkranz, J., Scheffer, T., and Spiliopoulou, M., editors, *Machine Learning: ECML 2006*, Lecture Notes in Computer Science, pages 282–293. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-46056-5.

Kocsis, L., Szepesvari, C., and Willemson, J. Improved Monte-Carlo Search. *Univ. Tartu, Estonia, Tech. Rep 1*, 2006.

Köstler, H. and Gmeiner, B. A Multi-objective Genetic Algorithm for Build Order Optimization in StarCraft II. *KI - Künstliche Intelligenz*, 27(3):221–233, August 2013. ISSN 0933-1875, 1610-1987. doi: 10.1007/s13218-013-0263-2.

Kovarsky, A. and Buro, M. Heuristic Search Applied to Abstract Combat Games. In Kégl, B. and Lapalme, G., editors, *Advances in Artificial Intelligence*, Lecture Notes in Computer Science, pages 66–78. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-31952-8.

Laird, J. and VanLent, M. Human-Level AI's Killer Application: Interactive Computer Games. *AI Magazine*, 22(2):15–15, June 2001. ISSN 2371-9621. doi: 10.1609/aimag.v22i2.1558.

Lanchester, F. W. *Aircraft in Warfare: The Dawn of the Fourth Arm.* Constable limited, 1916. ISBN 0-598-85489-4.

Lelis, L. H. S. Stratified Strategy Selection for Unit Control in Real-Time Strategy Games. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*, pages 3735–3741, Melbourne, Australia, August 2017. International Joint Conferences on Artificial Intelligence Organization. ISBN 978-0-9992411-0-3. doi: 10.24963/ijcai.2017/522.

Lim, Y. J. and Lee, W. S. Properties of forward pruning in game-tree search. In *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 2*, AAAI'06, pages 1020–1025, Boston, Massachusetts, July 2006. AAAI Press. ISBN 978-1-57735-281-5.

Louis, S. J. and Liu, S. Multi-Objective Evolution for 3D RTS Micro. In *2018 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8, July 2018. doi: 10.1109/CEC.2018.8477926.

Lucas, S. M. and Kendall, G. Evolutionary computation and games. *IEEE Computational Intelligence Magazine*, 1(1):10–18, February 2006. ISSN 1556-6048. doi: 10.1109/MCI.2006.1597057.

Mariño, J. R. H., Moraes, R. O., Toledo, C., and Lelis, L. H. S. Evolving Action Abstractions for Real-Time Planning in Extensive-Form Games. In *Proceedings of the Conference on Artificial Intelligence (AAAI)*, 2018.

Marsland, T. A. A Review of Game-Tree Pruning. *ICGA Journal*, 9(1):3–19, January 1986. ISSN 1389-6911. doi: 10.3233/ICG-1986-9102.

Miller, B. L. and Goldberg, D. E. Genetic Algorithms, Tournament Selection, and the Effects of Noise. *Complex Systems*, 9:193–212, 1995.

Millington, I. *AI for Games.* Taylor & Francis, a CRC title, part of the Taylor & Francis imprint, a member of the Taylor & Francis Group, the academic division of T&F Informa, plc, Boca Raton, third edition edition, 2019. ISBN 978-1-138-48397-2.

Minotti, M. The history of MOBAs: From mod to sensation, September 2014.

Mora, A. M., Fernández-Ares, A., Merelo-Guervós, J.-J., and García-Sánchez, P. Dealing with Noisy Fitness in the Design of a RTS Game Bot. In Di Chio, C., Agapitos, A., Cagnoni, S., Cotta, C., de Vega, F. F., Di Caro, G. A., Drechsler, R., Ekárt, A., Esparcia-Alcázar, A. I., Farooq, M., Langdon, W. B., Merelo-Guervós, J. J., Preuss, M., Richter, H., Silva, S., Simões, A., Squillero, G., Tarantino, E., Tettamanzi, A. G. B., Togelius, J., Urquhart, N., Uyar, A. Ş., and Yannakakis, G. N., editors, *Applications of Evolutionary Computation*, Lecture Notes in Computer Science, pages 234–244, Berlin, Heidelberg, 2012. Springer. ISBN 978-3-642-29178-4. doi: 10.1007/978-3-642-29178-4_24.

Moraes, R. O. and Lelis, L. H. S. Asymmetric Action Abstractions for Multi-Unit Control in Adversarial Real-Time

Games. In *The Thirty-Second AAAI Conference on Artificial Intelligence AAAI-18*, pages 876–883, 2018a.

Moraes, R. O. and Lelis, L. H. S. Nested-Greedy Search for Adversarial Real-Time Games. In *AIIDE*, 2018b.

Moraes, R. O., Mariño, J. R. H., Lelis, L. H. S., and Nascimento, M. A. Action Abstractions for Combinatorial Multi-Armed Bandit Tree Search. In *AAAI Publications, Fourteenth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2018.

Moss, R. Build, gather, brawl, repeat: The history of real-time strategy games. *Ars Technica*, September 2017.

Neufeld, X., Mostaghim, S., and Perez-Liebana, D. Evolving Game State Evaluation Functions for a Hybrid Planning Approach. In *IEEE Conference on Games 2019*, August 2019a.

Neufeld, X., Mostaghim, S., and Perez-Liebana, D. A Hybrid Planning and Execution Approach Through HTN and MCTS. In *The 3rd Workshop on Integrated Planning, Acting, and Execution - ICAPS'19*, pages 37–45, July 2019b.

O'Connor, A. Blitzkrieg 3 claims world's first RTS neural net, Boris. *Rock, Paper, Shotgun*, February 2017.

Ontañón, S. Experiments with Game Tree Search in Real-Time Strategy Games. *arXiv:1208.1940 [cs]*, August 2012.

Ontañón, S. The Combinatorial Multi-Armed Bandit Problem and Its Application to Real-Time Strategy Games. In *Proceedings of the Ninth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pages 58–64, 2013.

Ontañón, S. Informed Monte Carlo Tree Search for Real-Time Strategy games. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, Santorini, Greece, September 2016. IEEE. ISBN 978-1-5090-1883-3. doi: 10.1109/CIG.2016.7860394.

Ontañón, S. Combinatorial Multi-armed Bandits for Real-Time Strategy Games. *Journal of Artificial Intelligence Research*, 58:665–702, March 2017. ISSN 1076-9757. doi: 10.1613/jair.5398.

Ontañón, S. and Buro, M. Adversarial Hierarchical-Task Network Planning for Complex Real-Time Games. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI 2015)*, pages 1652–1658, 2015.

Ontañón, S., Synnaeve, G., Uriarte, A., Richoux, F., Churchill, D., and Preuss, M. A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft. *IEEE Transactions on Computational Intelligence and AI in Games*, 5(4):293–311, December 2013. ISSN 1943-068X, 1943-0698. doi: 10.1109/TCIAIG.2013.2286295.

Ontañón, S., Synnaeve, G., Uriarte, A., Richoux, F., Churchill, D., and Preuss, M. RTS AI Problems and Techniques. In Lee, N., editor, *Encyclopedia of Computer Graphics and Games*, pages 1–12. Springer International Publishing, Cham, 2015. ISBN 978-3-319-08234-9. doi: 10.1007/978-3-319-08234-9_17-1.

Ontañón, S., Barriga, N. A., Silva, C. R., Moraes, R. O., and Lelis, L. H. S. The First microRTS Artificial Intelligence Competition. *AI Magazine*, 39(1):75–83, March 2018. ISSN 2371-9621. doi: 10.1609/aimag.v39i1.2777.

Ouessai, A., Salem, M., and Mora, A. M. Improving the Performance of MCTS-Based µRTS Agents Through Move Pruning. In *2020 IEEE Conference on Games (CoG)*, pages 708–715, Osaka, Japan, August 2020a. IEEE. doi: 10.1109/CoG47356.2020.9231715.

Ouessai, A., Salem, M., and Mora, A. M. Online Adversarial Planning in μRTS : A Survey. In *2019 International Conference on Theoretical and Applicative Aspects of Computer Science (ICTAACS)*, Skikda, Algeria, December 2019. IEEE. doi: 10.1109/ICTAACS48474.2019.8988124.

Ouessai, A., Salem, M., and Mora, A. M. Parametric Action Pre-Selection for MCTS in Real-Time Strategy Games. In *VI Congress of the Spanish Society for Video Game Sciences*, pages 104–115, Madrid, Spain, October 2020b. CEUR-WS.

Ouessai, A., Salem, M., and Mora, A. M. Evolving action pre-selection parameters for MCTS in real-time strategy games. *Entertainment Computing*, 42:100493, May 2022. ISSN 1875-9521. doi: 10.1016/j.entcom.2022.100493.

Perez, D., Samothrakis, S., Lucas, S., and Rohlfshagen, P. Rolling horizon evolution versus tree search for navigation in single-player real-time games. In *Proceeding of the Fifteenth Annual Conference on Genetic and Evolutionary Computation Conference - GECCO '13*, page 351, Amsterdam, The Netherlands, 2013. ACM Press. ISBN 978-1-4503-1963-8. doi: 10.1145/2463372.2463413.

Perkins, L. Terrain analysis in real-time strategy games: An integrated approach to choke point detection and region decomposition. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 6, pages 168–173, 2010. ISBN 2334-0924.

Ponsen, M. J. V., Muñoz-Avila, H., Spronck, P., and Aha, D. W. Automatically Acquiring Domain Knowledge For Adaptive Game AI Using Evolutionary Learning. In *AAAI*, 2005.

Quiggin, J. *Generalized Expected Utility Theory: The Rank-Dependent Model*. Springer Science & Business Media, 1993. ISBN 978-94-011-2182-8.

Rabin, S. *AI Game Programming Wisdom*. Charles River Media, Hingham, Mass., 2007. ISBN 978-1-58450-077-3.

Richoux, F. Terrain Analysis in StarCraft 1 and 2 as Combinatorial Optimization. *arXiv preprint arXiv:2205.08683*, 2022.

Risi, S. and Togelius, J. Neuroevolution in Games: State of the Art and Open Challenges. *IEEE Transactions on Computational Intelligence and AI in Games*, 9(1):25–41, March 2017. ISSN 1943-0698. doi: 10.1109/TCIAIG.2015.2494596.

Robertson, G. and Watson, I. A Review of Real-Time Strategy Game AI. *AI Magazine*, 35(4):75–104, December 2014. ISSN 0738-4602, 0738-4602. doi: 10.1609/aimag.v35i4.2478.

Russell, S. J., Norvig, P., and Davis, E. Making Complex Decisions (Ch. 17). In *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd edition, 2010. ISBN 978-0-13-604259-4.

Schaeffer, J., Burch, N., Björnsson, Y., Kishimoto, A., Müller, M., Lake, R., Lu, P., and Sutphen, S. Checkers Is Solved. *Science*, 317(5844):1518–1522, September 2007. ISSN 0036-8075, 1095-9203. doi: 10.1126/science.1144079.

Schwab, B. *AI Game Engine Programming 2nd*. Course Technology, Cengage Learning, Boston, MA, 2nd ed edition, 2009. ISBN 978-1-58450-572-3.

Sephton, N., Cowling, P. I., Powley, E., and Slaven, N. H. Heuristic move pruning in Monte Carlo Tree Search for the strategic card game Lords of War. In *2014 IEEE Conference on Computational Intelligence and Games*, Dortmund, Germany, August 2014. IEEE. ISBN 978-1-4799-3547-5. doi: 10.1109/CIG.2014.6932892.

Shannon, C. E. XXII. Programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41(314):256–275, 1950.

Sharkey, S. Hail to the Duke. *1UP.com*, September 2004.

Shleyfman, A., Komenda, A., and Domshlak, C. On Combinatorial Actions and CMABs with Linear Side Information. In *ECAI 2014: 21st European Conference on Artificial Intelligence*, pages 825–830, 2014.

Silva, C. R., Moraes, R. O., Lelis, L. H. S., and Gal, K. Strategy Generation for Multi-Unit Real-Time Games via Voting. *IEEE Transactions on Games*, 2018. ISSN 2475-1502, 2475-1510. doi: 10.1109/TG.2018.2848913.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016. ISSN 0028-0836, 1476-4687. doi: 10.1038/nature16961.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., and Hassabis, D. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144, December 2018. ISSN 0036-8075, 1095-9203. doi: 10.1126/science.aar6404.

Spronck, P., Ponsen, M., Sprinkhuizen-Kuyper, I., and Postma, E. Adaptive game AI with dynamic scripting. *Machine Learning*, 63(3):217–248, June 2006. ISSN 1573-0565. doi: 10.1007/s10994-006-6205-6.

Stanescu, M. Using Lanchester Attrition Laws for Combat Prediction in StarCraft. In *Proceedings, The Eleventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-15)*, pages 86–92, 2015.

Stanescu, M., Barriga, N. A., and Buro, M. Hierarchical Adversarial Search Applied to Real-Time Strategy Games. In *Proceedings of the Tenth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, pages 66–72, 2014.

Stanescu, M., Barriga, N. A., Hess, A., and Buro, M. Evaluating real-time strategy game states using convolutional neural networks. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, Santorini, Greece, September 2016. IEEE. ISBN 978-1-5090-1883-3. doi: 10.1109/CIG.2016.7860439.

Steedman, M. *The Syntactic Process*. Language, Speech, and Communication. MIT Press, Cambridge, Mass, 2000. ISBN 978-0-262-19420-4.

Sun, L., Jiao, P., Xu, K., Yin, Q., and Zha, Y. Modified Adversarial Hierarchical Task Network Planning in Real-Time Strategy Games. *Applied Sciences*, 7(9):872, August 2017. ISSN 2076-3417. doi: 10.3390/app7090872.

Sun, P., Sun, X., Han, L., Xiong, J., Wang, Q., Li, B., Zheng, Y., Liu, J., Liu, Y., Liu, H., and Zhang, T. TStarBots: Defeating the Cheating Level Builtin AI in StarCraft II in the Full Game. *arXiv:1809.07193 [cs]*, December 2018.

Sutton, R. S. and Barto, A. G. *Reinforcement Learning: An Introduction*. MIT Press, 2018. ISBN 0-262-35270-2.

Tang, Z., Zhao, D., Zhu, Y., and Guo, P. Reinforcement Learning for Build-Order Production in StarCraft II. In *2018 Eighth International Conference on Information Science and Technology (ICIST)*, pages 153–158, June 2018. doi: 10.1109/ICIST.2018.8426160.

Tavares, A. R. and Chaimowicz, L. Tabular Reinforcement Learning in Real-Time Strategy Games via Options. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, 2018.

Tian, Y., Gong, Q., Shang, W., Wu, Y., and Zitnick, C. L. ELF: An Extensive, Lightweight and Flexible Research Platform for Real-time Strategy Games. In *31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.*, 2017.

Turing, A. M. Digital computers applied to games. *Faster than Thought*, 101, 1953.

Uriarte, A. and Ontañón, S. High-level Representations for Game-Tree Search in RTS Games. In *Tenth Artificial Intelligence and Interactive Digital Entertainment Conference, Artificial Intelligence in Adversarial Real-Time Games Workshop*, pages 14–18, 2014.

Uriarte, A. and Ontañón, S. Single believe state generation for partially observable real-time strategy games. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 296–303, New York, NY, USA, August 2017. IEEE. ISBN 978-1-5386-3233-8. doi: 10.1109/CIG.2017.8080449.

Vinyals, O., Ewalds, T., Bartunov, S., Georgiev, P., Vezhnevets, A. S., Yeo, M., Makhzani, A., Küttler, H., Agapiou, J., Schrittwieser, J., Quan, J., Gaffney, S., Petersen, S., Simonyan, K., Schaul, T., van Hasselt, H., Silver, D., Lillicrap, T., Calderone, K., Keet, P., Brunasso, A., Lawrence, D., Ekermo, A., Repp, J., and Tsing, R. StarCraft II: A New Challenge for Reinforcement Learning. *arXiv:1708.04782 [cs]*, August 2017.

Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., Oh, J., Horgan, D., Kroiss, M., Danihelka, I., Huang, A., Sifre, L., Cai, T., Agapiou, J. P., Jaderberg, M., Vezhnevets, A. S., Leblond, R., Pohlen, T., Dalibard, V., Budden, D., Sulsky, Y., Molloy, J., Paine, T. L., Gulcehre, C., Wang, Z., Pfaff, T., Wu, Y., Ring, R., Yogatama, D., Wünsch, D., McKinney, K., Smith, O., Schaul, T., Lillicrap, T., Kavukcuoglu, K., Hassabis, D., Apps, C., and Silver, D. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, November 2019. ISSN 1476-4687. doi: 10.1038/s41586-019-1724-z.

Walker, M. H. Strategy Gaming: Part I – A Primer. *GameSpy Articles*, 2002.

Wang, C., Chen, P., Li, Y., Holmgard, C., and Togelius, J. Portfolio Online Evolution in StarCraft. In *Proceedings, The Twelfth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-16)*, pages 114–120, 2016.

Weber, B., Mateas, M., and Jhala, A. A Particle Model for State Estimation in Real-Time Strategy Games. In *Seventh Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE'11)*, pages 103–108, 2011.

Yang, W., Zhang, Q., and Peng, Y. A Dynamic Hierarchical Evaluating Network for Real-Time Strategy Games. In Bevrani, H. and Shuhui, W., editors, *MATEC Web of Conferences*, volume 208, page 05003, September 2018. doi: 10.1051/matecconf/201820805003.

Yang, W., Xie, X., and Peng, Y. Fuzzy Theory Based Single Belief State Generation for Partially Observable Real-Time Strategy Games. *IEEE Access*, 7:79320–79330, June 2019. ISSN 2169-3536. doi: 10.1109/ACCESS.2019.2923419.

Yang, Z. and Ontañón, S. Learning Map-Independent Evaluation Functions for Real-Time Strategy Games. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, August 2018. doi: 10.1109/CIG.2018.8490369.

Yang, Z. and Ontañón, S. Extracting Policies from Replays to Improve MCTS in Real Time Strategy Games. In *The 2nd Workshop on Knowledge Extraction from Games Co-Located with 33rd AAAI Conference on Artificial Intelligence (AAAI 2019)*, Honolulu, Hawaii, 2019a.

Yang, Z. and Ontañón, S. 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, October 2019b.

Yang, Z. and Ontañón, S. Integrating Search and Scripts for Real-Time Strategy Games: An Empirical Survey. In *AAAI-20 Workshop on Reinforcement Learning in Games*, New York, 2020.

Yannakakis, G. N. and Togelius, J. *Artificial Intelligence and Games*. Springer International Publishing, Cham, 2018. ISBN 978-3-319-63518-7 978-3-319-63519-4. doi: 10.1007/978-3-319-63519-4.

Zhen, J. S. and Watson, I. Neuroevolution for Micromanagement in the Real-Time Strategy Game Starcraft: Brood War. In Cranefield, S. and Nayak, A., editors, *AI 2013: Advances in Artificial Intelligence*, Lecture Notes in Computer Science, pages 259–270, Cham, 2013. Springer International Publishing. ISBN 978-3-319-03680-9. doi: 10.1007/978-3-319-03680-9_28.

Abstract

Real-Time Strategy (RTS) games impose multiple complex challenges to autonomous game-playing agents (a.k.a. bots), that also relate to real-world problems. The real-time aspect and the astronomical size of the decision and state spaces of an RTS game overwhelm the usual search algorithms. Monte-Carlo Tree Search (MCTS) was successfully applied in games featuring large decision and state spaces, such as Go, and was able to attain super-human performance in agents like AlphaGo and AlphaZero. Thus, researchers turned to MCTS as a potential candidate for solving RTS Games, and several RTS-specific enhancements were implemented, such as the support for real-time progression and combinatorial decisions. Nevertheless, MCTS is still far from replicating its Go success in RTS games. In this thesis, we propose several approaches to ease the RTS dimensionality burden on MCTS, in hopes of finding a path towards higher performance. To this end, we have made use of a detrimental-move pruning approach, proposed an integrated action/state abstraction process, and optimized its parameters through an Evolutionary Algorithm (EA). These approaches were tested and validated in the µRTS research platform, and the results showed moderate to significant performance gains. We expect the proposed approaches could be applied in commercial RTS games in the near future.

Keywords: Real-Time Strategy Games, Monte Carlo Tree Search, Move Pruning, Action Abstraction, Parameter Optimization, Genetic Algorithms

ملـخــص

إن ألعاب الاستراتيجية ذات الوقت الفعلي تفرض تحديات معقدة على برامج الذكاء الاصطناعي التي تحاول لعبها. هذه التحديات مرتبطة بشكل وثيق بمشاكل واقعية مثل صفة الوقت الفعلي، و ضخامة فضاءي القرار و الحال اللتان تعيقان خوارزميات البحث المعروفة. إن خوارزمية مونتي كارلو للبحث الشجري تمكنت من إيجاد حل لألعاب معقدة، مثل لعبة ڤو المتميزة بفضاء كبير للقرار و الحال. ذلك ما لفت انتباه الباحثين، و دفعهم لتكييف هذه الخوارزمية لأجل ألعاب الاستراتيجية ذات الوقت الفعلي. الشيء الذي أدى لظهور العديد من التحسينات الخاصة، كدعم الوقت الفعلي و فضاء القرار المركب. لكن بالرغم من ذلك لا تزال خوارزمية مونتي كارلو للبحث الشجري غير قادرة على تجسيد نجاحها في لعبة ڤو في ألعاب الاستراتيجية ذات الوقت الفعلي. من خلال هذه الأطروحة، نحاول تقديم بعض الطرق التي قد تفيد في تخفيف عبء ضخامة فضاءات البحث على خوارزمية مونتي كارلو للبحث الشجري، حيث قمنا باستعمال طريقة لتشذيب القرارات المضرة، ثم قدمنا اطارا تجريديا لفضاءي القرار و الحال، و قمنا بتحسين معايير هذا الاطار من خلال خوارزمية تطورية. النتائج المتحصل عليها بعد التجارب في منصة بحث خاصة بالألعاب الاستراتيجية ذات الوقت الفعلي، أظهرت تحسنا في الأداء بنسب تتفاوت بين المتوسطة و المعتبرة. نتوقع امكانية تطبيق هاته المقاربات في ألعاب الاستراتيجية ذات الوقت الفعلي التجارية في المستقبل القريب.

الكلمات المفتاحية: ألعاب الاستراتيجية ذات الوقت الفعلي، مونتي كارلو للبحث الشجري، تشذيب القرارات، تحسين المعايير ، الخوارزميات الجينية

Résumé

Les jeux de stratégie en temps-réel (RTS) posent des défis considérables à l'encontre des agents autonomes (dites aussi, "bots"), des défis étroitement liés aux problèmes du monde réel. L'aspect temps-réel, et les énormes espaces d'état et de décision, accablent les algorithmes de recherche traditionnelle. L'algorithme Monte-Carlo Tree Search (MCTS) s'est réjoui d'un succès immense dans les domaines ayant de larges espaces d'état et de décision, tel que le jeu Go, où il a pu démontrer des capacités surhumaines à travers les agents AlphaGo et AlphaZero. Les chercheurs ont pris note, et ont visé à adapter MCTS pour les jeux RTS. Diverses améliorations spécifiques aux jeux RTS ont vu le jour, comme le support de l'aspect temps-réel et des décisions combinatoires. Néanmoins, MCTS reste incapable de reproduire son succès dans Go dans les jeux RTS. Dans cette thèse, on propose des approches qui tentent de diminuer l'impact de la haute dimensionalité sur MCTS, visant à atteindre des performances plus élevées dans le domaine des RTS. Pour cela, on a utilisé une méthode d'élagage des actions, et on a proposé un mécanisme d'abstraction d'états et d'actions intégré, qu'on a ensuite optimisé par un algorithme évolutionnaire. Ces approches ont été testées et validées dans la plateforme de recherche µRTS, et les résultats obtenus démontrent des gains de performance de degré modéré jusqu'à un degré considérable. On prévoit que les approches proposées pourraient être appliquées dans les jeux RTS commerciaux dans un proche avenir.

Mots-clés: Jeux de stratégie en temps réel, recherche d'arbre de Monte Carlo, élagage d'actions, abstraction des actions, optimisation des paramètres, algorithmes génétiques