Outcome Prediction and Hierarchical Models in Real-Time Strategy Games

by

Adrian Marius Stanescu

A thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computing Science

University of Alberta

Abstract

For many years, traditional boardgames such as Chess, Checkers or Go have been the standard environments to test new Artificial Intelligence (AI) algorithms for achieving robust game-playing agents capable of defeating the best human players. Presently, the focus has shifted towards games that offer even larger action and state spaces, such as Atari and other video games. With a unique combination of strategic thinking and fine-grained tactical combat management, Real-Time Strategy (RTS) games have emerged as one of the most popular and challenging research environments. Besides state space complexity, RTS properties such as simultaneous actions, partial observability and real-time computing constraints make them an excellent testbed for decision making algorithms under dynamic conditions.

This thesis makes contributions towards achieving human-level AI in these complex games. Specifically, we focus on learning, using abstractions and performing adversarial search in real-time domains with extremely large action and state spaces, for which forward models might not be available.

We present two abstract models for combat outcome prediction that are accurate while reasonably computationally inexpensive. These models can inform high level strategic decisions such as when to force or avoid fighting or be used as evaluation functions for look-ahead search algorithms. In both cases we obtained stronger results compared to at the time state-of-the-art heuristics. We introduce two approaches to designing adversarial look-ahead search algorithms that are based on abstractions to reduce the search complexity. Firstly, *Hierarchical Adversarial Search* uses multiple search layers that work at different abstraction levels to decompose the original problem. Secondly, *Puppet Search* methods use configurable scripts as an action abstraction mech-

anism and offer more design flexibility and control. Both methods show similar performance compared to top scripted and state-of-the-art search based agents in small maps, while outperforming them on larger ones. We show how to use Convolutional Neural Networks (CNNs) to effectively improve spatial awareness and evaluate game outcomes more accurately than our previous combat models. When incorporated into adversarial look-ahead search algorithms, this evaluation function increased their playing strength considerably.

In these complex domains forward models might be very slow or even unavailable, which makes search methods more difficult to use. We show how policy networks can be used to mimic our *Puppet Search* algorithm and to bypass the need of a forward model during gameplay. We combine the much faster resulting method with other search-based tactical algorithms to produce RTS game playing agents that are stronger than state-of-the-art algorithms. We then describe how to eliminate the need for simulators or forward models entirely by using Reinforcement Learning (RL) to learn autonomous, self-improving behaviors. The resulting agents defeated the built-in AI convincingly and showed complex cooperative behaviors in small scale scenarios of a fully fledged RTS game.

Finally, learning becomes more difficult when controlling increasing numbers of agents. We introduce a new approach that uses CNNs to produce a spatial decomposition mechanism and makes credit assignment from a single team reward signal more tractable. Applied to a standard Q-learning method, this approach resulted in increased performance over the original algorithm in both small and large scale scenarios.

Acknowledgements

I would first like to thank my supervisor Michael Buro for his excellent guidance and support throughout my doctoral studies and for his help with less academic ventures such as becoming a prize-winning Skat player and a passable skier.

I would also like to thank my committee members Vadim Bulitko and Jonathan Schaeffer for their comments and feedback that helped improve my work. Thanks to Dave Churchill for paving the way and helping me get started. Special thanks to Nicolas Barriga for all the coffee and moral support shared before deadlines as well as the pizza, movie and scotch afterwards.

Thanks to my roommates Sergio, Mona and Graham for letting me win at boardgames, sharing pancakes and countless other delicious meals. Aristotle for offering invaluable assistance in staying positive (while overspending on biking items) and Stuart for helping me stay fit.

I would especially like to thank my family. My wife, Raida has been extremely supportive throughout this entire adventure and has made countless sacrifices to help me get to this point. My mom deserves many thanks for her infinite patience and implicit faith in my capabilities.

Table of Contents

List of Figures

Chapter 1

Introduction

Games are an attractive medium for exploring the capabilities of AI in constrained environments and fixed set of rules. As a result, problem-solving techniques can be developed and evaluated before being applied to more complex real-world problems [185]. The relation between AI research and games started in 1950 when Claude Shannon proposed a Chess playing algorithm just a few years after the first computer was built [192]. He explained that 'it is hoped that a satisfactory solution of this problem will act as a wedge in attacking other problems of a similar nature and of greater significance' and gave a few examples such as military decision making or even electronic design. Just one year later Alan Turing wrote the first Chess playing program but he did not have the hardware able to run it yet. Many other programs were developed in the following years, until 1957 when the first full-fledged game of Chess was played by a computer at MIT [20].

Since the 1950s, AI has been applied to board games such as Chess, Scrabble, Backgammon or Go, creating competition which has sped up the development of many heuristic based search techniques [185]. Modern game playing programs have been able to defeat top Chess and more recently, Go grandmasters. Moreover, some of these games have been declared nearly or completely solved (i.e., there are programs capable of playing optimally and neither humans nor other computer programs can perform better), for example Checkers [186], Connect-4 [229] and more recently Heads-up Limit Hold'em Poker [23].

Most of the algorithms used to solve boardgames are based on tree search methods with well designed evaluation functions, large databases of game instances and sometimes even special purpose hardware such as Deep Blue [32]. AlphaGo Zero [196] used an MCTS variant and neural networks to learn to play simply by playing games against itself, starting from completely random play. It was trained for 40 days using 64 GPU workers and 19 CPU parameter servers and reached a rating of over 5100 Elo, while no human Go player has ever been rated over 3700 Elo [1]. The hardware cost for a single AlphaGo

[1] `https://www.goratings.org/en/history/`

Zero system, including custom components, has been quoted as around US$25 million [76]. The rapidly improving hardware capabilities have (sometimes even single-handedly) helped computer programs to reach world-master performance in increasingly complex boardgames.

Over the past several years the focus has shifted towards research on video game AI, initiated by Laird and VanLent in 2001 with their call for using video games as a testbed for AI research [130]. The authors saw video games as a potential area for iterative advancement in increasingly complex scenarios. Just two years later Buro called for increased research in Real-Time Strategy (RTS) games, recognizing that they provide a perfect sandbox for exploring various complex challenges that are central to game AI (and many other problems) [27].

Motivation for RTS AI has since grown rapidly with the emergence of competitions such as the Open RTS (ORTS) AI competition [26, 29], the Google AI Challenges such as Planet Wars [80] and Ants [81]), and *StarCraft* AI Competitions (organized as part of AIIDE and CIG conferences or as independent tournaments [39]). Less than 20 years later, the biggest tech companies in the world such as Google [82], Facebook [61], Microsoft [145] or IBM [253] are using video games as means through which to challenge the state-of-the-art in AI.

1.1 Real Time Strategy Games

RTS games, while similar to traditional abstract strategy games such as Chess or Go, offer more complex environments in terms of state space size, the number of actions that can be performed at any time step, and the number of actions required to reach the end of a game. Each player observes the battlefield from a top-down perspective and controls multiple agents simultaneously in real time. The customary goal is to overcome opponents by destroying their armies and bases. To achieve this, gathering resources quickly and building a powerful military force are essential. Being able to build a superior economic base often helps to accomplish these tasks. RTS games demand precise handling of individual units (micro management) but also require planning in an abstract world view and deciding what to spend one's resources on: economy, army or researching new technology (macro management).

Different maps with thousands of possible unit positions, army compositions, sequences of building structures and researching technologies, and the resulting very large action space make RTS games much more complex than traditional games. The number of possible states in Chess is approximately 10^{50}, Go has around 10^{170} states, while StarCraft has at least 10^{1000} [166]. Brute force tree-search becomes impractical, and as a result, AI for RTS games usually exhibits some form of abstraction or simplification, generally coupled with a modular divide-and-conquer framework. Often, sub-problems are studied instead of learning to play RTS games end-to-end, and micro-management

scenarios are the most frequently tackled challenges [116].

1.1.1 Multiplayer Online Battle (MOBA) Arena Games

The MOBA game genre, also known as action real-time strategy (ARTS), has originated as a subgenre of RTS games and is undoubtedly one of the most popular in gaming today. Reports have estimated the number of monthly active players of the MOBA game *League of Legends* at 100 million players in August 2017, almost as many as all other well-known eSports combined (second and third place were for the first person shooter *Call of Duty* and the collectible card game *Hearthstone* with 28 and 23 million respectively, while another MOBA game – *Dota 2* – was in fourth with 12 million) [207].

In MOBA games players control a single character plus a few minions and compete in teams of usually 5 players each. The objective is to destroy the opposing team's main structure with the assistance of periodically spawned computer-controlled units that march forward along set paths. Characters typically have several abilities and various advantages each, that improve over the course of a game. Game mechanics usually do not include construction of buildings or unit training, but focus heavily on character development and customization through items and abilities.

Similarly to standard RTS games, there are complex decisions to be made in respect to resource management, progression and control of the player's character which combine both micro and macro levels of strategy. While RTS players focus more on the higher level strategies, MOBA games can offer more difficult micro challenges and additionally require teamwork and communication between players. This focus on multi-player strategies and interactions might be one of the reasons why MOBAs have pushed other genres aside to become the largest eSport in the world in terms of players [207], viewers [35] and prize money awarded [56]. In 2017, over US$113 million was awarded in prizes, from which over US$50 million was totaled by just two MOBA games – Dota 2 and League of Legends – while from the other genres the top games were the shooters Counter-Strike and Call of Duty at US$12 and US$4 million each [56]. StarCraft II was the top RTS game at slightly over US$3 million.

1.1.2 eSports and RTS Game AI

One incentive for AI research comes from the video game industry, which generated well over US$100bn in revenues in 2017 [16]. While the focus often falls on game aesthetics, a great interactive experience still plays an important part. Crucial to this experience are adaptive AI-driven agents, which can interact with the player in a variety of roles such as opponents, teammates or other non-player characters (NPCs). In addition, besides providing a great experience for players, eSports bring an increasing need for AI-based tools for engagement with spectators, commentators and game developers which now

have to be even more concerned by game balancing.

The increased popularity of eSports [208] brings extra incentive to AI development. The established industry and competitive gaming scene provides dependable access to professional human players, which offer a reliable way of evaluating research progress. In contrast, games played only at amateur level might seem reasonably difficult at first but may lack real challenges. For example, Arimaa [214] was designed to be difficult to play by computers while easy to master by humans. Eventually, computer programs were able to defeat the best human players using only techniques already employed in Chess and Go playing programs. However, after ten years of research it is hard to tell how strong those computer programs really are, as there are no expert players to judge their progress against. The large community offers great insight, and we are now much more aware of the capabilities and limitations of current efforts in RTS or MOBA game AI. A good example is the discussion around the recent performance of OpenAI Five during the latest Dota 2 major competition in August 2018 [167]. Analysis of the the two games – defeats by a top 18 Dota 2 team and a team of Chinese superstar players – while of average length and apparently competitive, showed that the AI excels at short-term, reactionary back-and-forth teamfights but lacks high level strategic thinking required to come back from behind or seal a victory.

Another advantage brought by the popularity of RTS and MOBA games is that they provide one element that many machine learning systems need and can benefit from: data. Most games publish complete replay data of matches, and there are a significant number of parsing and analyzing tools that enable hobbyists and researchers to use this public data for their own interests. Benchmarking on publicly available datasets will only help progression of AI methods, along with being able to play versus professionals and insightful commentary and analysis from experts and community members.

1.2 Challenges

RTS games pose challenging problems for AI development and research. The need to manage resources, to make critical decisions in situations while uncertain about the current strength in relation to opposing forces, to conduct spatial and temporal reasoning and the agent co-ordination required in this genre still presents challenges to even the most state-of-the-art techniques. Besides state space complexity, playing RTS games successfully – compared to traditional turn-based board games – is a more difficult task due to the following reasons [166]:

- In RTS games, players act simultaneously (i.e., more than one player can issue commands at any given time).
- Gameplay takes place in real time. Players do not need to wait for their

opponents to issue the next move, and only have a limited amount of time to react to events on the playing field.

- The battlefield is usually partially observable, covered by the so called fog of war. Visibility is restricted to areas close to the player's own structures and units, and the map needs to be constantly explored (process called scouting).
- Some RTS games are non-deterministic, in the sense that some actions may fail or have uncertain outcomes, such as probabilistic critical strikes or missing when shooting an enemy uphill.
- RTS game maps are dynamic. Resources on the map may be depleted or static objects may be destroyed to create new paths.

Recent research has brought to attention a few additional challenges, relevant to current state-of-the-art techniques, such as how to best use the large number of game replays available. Current challenges in RTS game AI have been grouped in five main different areas [166, 230]:

1. Adversarial Planning in Large Real-Time Domains Standard adversarial game tree search approaches are not directly applicable to RTS games due to the large state and action spaces. Possible approaches include division into smaller sub-problems for which search-based methods are feasible, such as high level objectives like choosing whether to improve one's economy or to build a larger army, or lower level goals such as coordinating one's troops in a small region of the map. Both spatial and temporal reasoning are important factors, and *when* or *where* are as important as *what* strategy to implement. If the game engine cannot act as a forward model, or there is no access to one, such a model would have to implemented at the representation level used by the planning method. For RTS games, modeling combat between large armies has significant impact on high level decisions, and is of particular interest.

2. Learning Most learning approaches in RTS games have been used for addressing sub-problems only, and can be organized in three types:

- *Prior Learning*: typically uses large numbers of replays or other acquired game data to infer some form of knowledge offline before starting to play, for example, predicting the next expert human move in AlphaGo [195].
- *In-Game Learning*: concentrates on evolving new strategies or improving the current ones while playing the game, by adapting to the opponent. Reinforcement learning techniques can be used as well, but usually evaluation is done using a small number of games – for example tournaments – and nowhere near what is required by common deep learning methods. This is related to the more general few-shot learning and transfer learning challenges.

- *Inter-Game Learning*: is usually done offline, and focuses on using knowledge from past games to progress in the next one. For example, game-theoretical concepts have been used to model the strategy selection process in StarCraft as a metagame [224]. Some state-of-the art StarCraft bots use bandit based Monte-Carlo planning to choose strategies according to past performance against their opponent [166].

3. Spatial Reasoning When terrain and geographical factors impact decisions, spatial reasoning is required to detect, exploit or estimate the effect of these properties. In RTS games, it is needed at the long term planning level for tasks such as building placement or choosing base expansion locations. At the shorter term level, there are tactical decisions such as fleeing or reinforcing contested areas, creating and avoiding ambushes and using terrain features such high ground and choke-points to obtain advantages in combat.

4. Imperfect Information As mentioned above, in most RTS games players cannot observe the whole map, and everything that is outside the sight range of one's units is covered by the fog of war. Knowledge can be gained via efficient scouting, and learning methods or inference modules can use past or present information about the opponent to make assumptions about what is possible, or probable with respect to his assets or strategy.

5. Task Decomposition For all the reasons already mentioned in this section, RTS games are difficult to solve end-to-end, and most existing methods are based on decomposition into smaller sub-problems, which they address jointly or independently. Based on both time scale and level of abstraction, these sub-problems have been classified in three categories in both literature from military command [248] and AI research [37]: *Strategy*, *Tactics* and *Reactive Control*. These categories imitate a military command hierarchy in both use of abstraction and its flow of information amongst the different commanders. For example, a high level commander can make a broad global strategic decision based on the assembled knowledge of enemy and own resources and map positioning. Based on this decision, tactical orders can be given to lower level commanders that only have access to local information required to achieve the given goal.

Related to both task decomposition and learning is the multi-agent aspect of RTS games. Current multi-agent learning algorithms often use assumptions of independence between the agents to avoid large action spaces. Scaling effectively towards similar levels of performance as existing single agents algorithms is a popular and open challenge.

These problems can also be found in real life situations such as air traffic controlling, automated vehicles, drone exploration and military analysis, and even weather prediction. Most areas with partial information and dynamic environments can benefit from using tactical analysis techniques developed for

RTS games. Games are a good testbed for AI development, and can be seen as a limited environment (due to their finite discrete simplifications and well defined rules) that can nonetheless model almost all characteristics of the real world. Developing and testing AI algorithms in game environments benefits research by greatly reducing the costs of real world resources and decreasing testing effort.

1.3 Contributions

The work presented in this dissertation tackles some of the challenges described in the previous sections. More specifically, we make the following contributions:

- Chapter 3: We describe a methodology for predicting the most probable army composition produced by an opponent in imperfect information strategy games. This system is based on a declarative knowledge representation paradigm, and thus domain specific knowledge can easily be encoded for any given adversarial environment. Predictions are very fast and thus the algorithm would be a useful component in any complex real-time domain with hidden opponent actions.

- Chapter 4 and 5: We design a combat model which approximates units' strength from a small number of past battles. It accurately predicts both battle outcomes as well as what type and how many troops are needed to defeat any given army. Extensions are implemented in a second combat model, which is both faster and more accurate, and can predict the remaining army sizes and thus be used as a forward model. Such models can inform high level strategic decisions such as when to force or avoid fighting or be used as evaluation functions for look-ahead search algorithms. In both cases we obtained stronger results compared to then state-of-the-art heuristics.

- Chapter 6: We propose an adversarial search algorithm with multiple search layers that work at different abstraction levels. Hierarchical decomposition is used to provide the layers with partial or abstract views of the game state and tasks to accomplish, in order to reduce the overall branching factor of the search. This approach is useful for adversarial domains with many agents that can be decomposed in relatively independent sub-problems.

- Chapter 8: We propose an alternative adversarial look-ahead search framework in which abstractions are easier to design and implement compared to the one introduced in Chapter 6. Configurable scripts with choice points are used as an abstraction mechanism, and we show that very strong agents can be built even from basic scripts if we use accurate

evaluation functions. We show how to combine this method with other search-based tactical algorithms to produce strong RTS game playing agents, which won the 2017 μRTS tournament.

- Chapter 7: We design state evaluation functions based on deep Convolutional Neural Networks (CNNs) that consider the entire game state, and are not restricted to tactical situations. We showed accuracy improvements over other state-of-the-art evaluations and considerably stronger gameplay when incorporated into adversarial look-ahead search algorithms. This approach requires very little domain knowledge compared to combat-specific models, and can be easily applied to other grid-like domains.

- Chapter 9: We describe how CNNs and Reinforcement Learning (RL) can be used to learn autonomous, adaptive and self-improving behavior in complex cooperative-competitive multi-agent environments. No simulators or forward models are required, and agents defeated the built-in AI and showed complex cooperative behaviors in a AAA game [2].

- Chapter 10: We design a novel learning approach for cooperative-competitive multi-agent RL problems when agents are learning from a single team reward signal. CNNs are used to produce a spatial decomposition mechanism which learns action-value functions on a per-sector basis instead of per-agent. We show that our new architecture made long-term credit assignment from a single team reward more tractable, and improved scaling to larger numbers of agents.

1.4 Publications

This article-based dissertation contains previously published material, and all chapters outlined above correspond to one paper of which I was the first author of. The exception is Chapter 8 which summarizes three papers to which I contributed as second author. The list of publications from this thesis are:

- (2013) Stanescu, Poo Hernandez, Erickson, Greiner and Buro. Predicting army combat outcomes in StarCraft [206]. *AIIDE*

- (2014) Stanescu, Barriga and Buro. Introducing Hierarchical Adversarial Search, a Scalable Search Procedure for Real-Time Strategy Games [201]. *ECAI poster*

[2] AAA is an informal classification used for video games produced and distributed by a mid-sized or major publisher, typically having higher development and marketing budgets. AAA game development is associated with high economic risk and with high levels of sales required to obtain profitability.

- (2014) Stanescu, Barriga and Buro. Hierarchical Adversarial Search Applied to Real-Time Strategy Games [200]. *AIIDE*
- (2015) Stanescu, Barriga and Buro. Using Lanchester Attrition Laws for Combat Prediction in StarCraft [202]. *AIIDE*
- (2016) Stanescu, Barriga, Hess and Buro. Evaluating Real-Time Strategy Game States Using Convolutional Neural Networks [204]. *CIG*
- (2016) Stanescu and Čertickỳ. Predicting Opponent's Production in Real-Time Strategy Games with Answer Set Programming [205]. *TCIAIG*
- (2018) Stanescu and Buro. Multi-Agent Action Network Learning Applied to RTS Game Combat. *AIIDE workshop*

- (2015) Barriga, Stanescu and Buro. Puppet Search: Enhancing Scripted Behavior by Look-Ahead Search with Applications to Real-Time Strategy Games [11]. *AIIDE*
- (2017) Barriga, Stanescu and Buro. Combining Strategic Learning and Tactical Search in Real-Time Strategy Games [12]. *AIIDE*
- (2017) Barriga, Stanescu and Buro. Game Tree Search Based on Non-Deterministic Action Scripts in Real-Time Strategy Games [14]. *TCIAIG*

Two other publications mentioned in the Appendix are:

- (2014) Barriga, Stanescu and Buro. Parallel UCT Search on GPUs [10]. *CIG*
- (2014) Barriga, Stanescu and Buro. Building Placement Optimization in Real-Time Strategy Games [9]. *AIIDE*

Chapter 2

Literature Survey

This chapter first mentions the most popular platforms for research applied to RTS games. As shown in Section 1.2, RTS games present many challenges and this chapter reviews existing research in this domain. The same decomposition presented above is used to organize this work, with a section dedicated to each of Reactive Control, Tactics and Strategy. Another section describes holistic approaches, which try to solve the whole problem using a single framework. A separate section is dedicated to deep reinforcement learning, as progress in this domain is more recent and follows a different trajectory that is better presented independently.

Due to the difficulty of implementing full, working algorithms for RTS games, a few research platforms have been developed. They offer more restricted scenarios and minimize the amount of engineering required to evaluate new algorithms. The most popular among RTS research are:

- *SparCraft* – a simplified StarCraft combat simulator that is much faster than the original engine and has been used for research on unit control (also see appendix B.2);
- *μRTS* – a less complex academic RTS game, deterministic, fully-observable, with simultaneous and durative actions (see appendix B.3). AI competitions have been organized recently as part of CIG 2017 and CIG 2018 conferences [163].

There are other research platforms that have seen less use, especially after the release of BWAPI in 2009 and μRTS in 2012: the academic RTS engine *ORTS* [26], the open source WarCraft II clone *Wargus* [3] and the 3D RTS game engine *SpringRTS* [2].

Successful RTS game AIs usually address many of the problems and challenges presented in Section 1.2 simultaneously. Hence, it would be difficult to label RTS AI research according to these challenges and researchers prefer a classification by abstraction levels [166, 176]. The chosen categories are inspired by military command hierarchies [248]: Strategy, Tactics and Reactive Control.

The highest level decisions, which have impact on a global scale and are concerned with winning the war, are part of one's ***strategy*** (e.g., build order, army compositions). It should be noted that here strategy is considered an abstraction category and is a different concept than strategy in a game theoretic context. Next, ***tactics*** has a smaller scale, both in time and spatially. Tactical problems often involve particular groups of units and smaller map areas, and usually aim to accomplish a specific objective such as winning a battle or capturing a certain location. Lastly ***reactive control*** finds low-level orders (such as move to position P or fire at enemy unit E) for individual units. While tactical decisions usually have a short-term time objective or plan, varying from seconds to minutes, reactive control decisions have millisecond granularity.

Often the line dividing these abstraction levels is not completely clear, and it is better to consider them as areas on a continuous scale. In the following three sections we will offer more detail and describe relevant research in each of these categories.

2.1 Reactive Control

This category encompasses the AI decisions with the shortest time horizon and mostly deals with actions at the unit level. Reactive control decisions usually are combat-related: how units will move, attack and use their abilities to defeat enemy squads or achieve other goals.

Influence Maps have been widely used to make combat decisions. They store relevant information about game objects such as units or wall as numerical influence, which is stronger closer to the specific object and degrades with distance. They are closely related to the idea of potential fields [89], sharing the same underlying principles. Influence maps have been used to avoid obstacles or to keep the optimum distance from opponents while shooting [231]. This behavior is called "kiting" and it is used by the StarCraft bot Nova [231]. Influence maps can also be used to implement more intelligent squad movement such as flanking [49], and can even help dealing with fog-of-war [88].

The main disadvantage of using influence maps is that many parameters need to be hand-tuned in order to obtain specific behaviors. Consequently, there has been research aimed at learning such parameters automatically, for example self organizing maps [173] or by using RL [140]. However, even these approaches have the drawback of occasionally leaving agents stuck in local optima.

Reinforcement Learning has been used to control units in RTS games, as it is relatively easy technique to apply to a new domain; it only requires a scenario description, a set of possible actions and a reward metric [142]. Different RL algorithms are compared, without finding significant differences,

in small scale StarCraft combat scenarios [245]. However, the large problem space of RTS games and the long delay of rewards make RL methods difficult to use for anything other than very short-term planning. The system proposed in [193] beats the built-in StarCraft AI in 3 unit fights, but fails to defeat it if the number of units is larger than 6 per side.

A hierarchical decomposition is proposed as a way of increasing the effectiveness of RL methods [143]. The game complexity is decreased using a hierarchical decomposition to combine units' Q-functions at the group (squad) level. Another idea to reduce the search space is learning one Q-function for all units of the same type [110]. Other recent RL research, based on advances in deep networks, is discussed in Section 2.5.

Game Tree Search techniques have been employed with success in small scale or abstracted instances of RTS combat scenarios. SparCraft was introduced to allow fast evaluation of unit actions [44]. It is a simulator that approximates StarCraft combat while ignoring some of the complex mechanisms such as unit collision, acceleration or casting spells. For more details on SparCraft, see Appendix B.2. The authors designed an alpha-beta search method that takes actions duration into account and found it to be effective for small and medium sized StarCraft combat instances.

The large number of actions and possible states, coupled with RTS time constraints of tens of milliseconds per search, makes this approach too slow for two player combat scenarios larger than 8 units per side. Portfolio Greedy Search is a search algorithm designed to work better in large-scale combat [43]. This algorithm limits the actions searched for each unit to actions produced by a set of scripts called a portfolio. Furthermore, it applies a hill-climbing technique to reduce the exponential number of unit actions combinations to a linear amount.

However, despite good performance within simulations, the results do not translate completely to the real StarCraft game. The inconsistency is caused by the discrepancy between action results in the simulation and the actual game. Researchers do not have access to the original StarCraft engine, and the outcome of actions issued through the BWAPI framework cannot currently be predicted with 100% certainty. More accurate simulators need to be implemented to overcome this issue, and this is an open research avenue.

2.2 Tactics

Tactical reasoning is a step up in scale from reactive control, and deals with decisions on a time scale of usually less than one minute. Tactics involves reasoning about the different abilities of units within a group and about spatial positioning of different units or groups of units on the battlefield. Typically, research in this category can be classified as either terrain analysis or decision

making [166].

Terrain Analysis assists decision making in games by providing map information in a systematical way. Wargames especially need qualitative spatial information [64]. Most StarCraft bots use the BWTA library, which implements relevant regions/choke points detection via Voronoi decomposition [172].

Another tactical decision that falls under terrain analysis is *building placement*. Defensive or unit training buildings can be placed near the opponent's base to support an aggressive strategy. Building placement also has an impact on the routes taken by units (e.g., one can block one's own base entrance to defend [34]). This paper only attempts to create a wall-type structure to help against early enemy attacks, and recognizes that other goals such as preventing enemy scouting or optimizing economic layout should also be pursued.

Decision Making *Scouting* the enemy is very important to both tactical and strategic decision making. However, most StarCraft bots assume perfect information, with a few exceptions [242]. The authors use particle filtering for predicting positions of previously seen enemy units in StarCraft, with decreasing confidence over time.

Once a player controls enough units, he must decide *when and where to attack* the opponent, taking into account the army compositions, technology levels and relevant spatial information. For example, in StarCraft two popular attack patterns are

- *timing attacks* – executed during a certain period in which an attack is stronger than outside that time window (e.g., own upgrade completed providing a sharp increase in strength but opponent's has not)
- attacking the opponent while he is building a new base, called expansion.

Decisions such as how to split one's army, group the units, flank or attack multiple areas at the same time are also taken at the tactical level. However, there is not much work done in this area, and most combat related research is focused in the lower level area of reactive control.

2.3 Strategy

Strategy encompasses the highest level decisions made during an RTS game. Strategic decisions are concerned with the long-term results of player actions. The following subsections describe the most important strategic sub-problems and relevant research.

Hard-coded and scripted methods (such as finite state machines [108]) that rely heavily on expert knowledge are the most common approaches in both research and commercial RTS AI. These methods are popular because they are

easy to design, computationally inexpensive, and simplify transferring expert domain knowledge into basic AI behavior. However, these approaches cannot adapt to circumstances unforeseen by the designers and exhibit static and easily identifiable behaviors. For that reason, human players can easily exploit and defeat such AIs.

Plan Recognition An important factor in strategic decision-making is being able to predict the opponent's strategy. This is a difficult task in RTS games due to not having perfect information about the enemy actions and units, concealed by fog-of-war. One of the first attempts to tackle this problem in StarCraft was by using recorded games (replays) to build a decision tree which is then used to predict the opponent's current strategy [240]. A similar idea uses a decision tree framework as well, but the tree structure is created manually using expert knowledge [118].

Alternatively, probabilistic plan recognition uses game replays to train opponent models automatically, without human input. The models learned can then be used to predict unobserved parts of the enemy's state, or future enemy actions given his past and current states. Hidden Markov Model (HMM) with state transitions happening every 30 seconds are used for prediction during the first 7 minutes of a StarCraft game [51]. This work is extended later to predict exact numbers of each unit and building type (every 30 seconds) instead of only whether it exists or not [107]. This new model takes scouting into account to infer if a unit/building was not discovered because it does not exist or because the player has not spent enough time searching for it.

A semi-supervised Bayesian model has been used instead of an HMM to predict opening strategies [216]. The authors also predict the current technology level of an opponent using an unsupervised Bayesian model in [217]. Their models are very robust providing predictions which on average are wrong by only one building after randomly hiding 80% of the enemy base. With perfect information they accurately predicted almost four buildings, on average, into the future. However, when using these systems in their 2012 StarCraft AI Competition bot, they only placed 4th.

Case-Based Reasoning (CBR) is also used for plan recognition, for example to try to predict the success of a specific plan in a particular situation by taking into account results from professional replays [109] . CBR was also used to accurately classify an opponent according to a few abstract labels such as defensive, aggressive, etc in the Spring RTS engine [184].

Planning systems are used to generate action sequences players should follow in order to achieve specific goals. Some popular approaches detailed below are Case-Based Planning (CBP), Goal-Driven methods and Hierarchical Planning.

Case-Based Planning Similarly to CBR, a CBP system aims to retrieve potential plans or plan fragments that are likely to prove useful in the current

context. For example, a real time case-based system was implemented to play Wargus (a Warcraft II clone) [165]. The authors learn plans from human demonstration available in the form of annotated game logs which contain sequences of actions for particular states and the goals the human was actively pursuing. Previous CBP work is later improved by using a decision tree to increase the plan retrieval's speed and quality [147]. However, significant effort is still needed for annotating logs.

Goal-driven Planning One limitation of CBP methods is that they do not actively try to reason about why a goal is succeeding or failing, but only choose a good plan for the current context. In comparison, in goal-driven models the agent reasons about its goals, identifies when they need to be updated, and adapts them as needed for subsequent planning and execution [152].

One such system was implemented for StarCraft [241]. The system maintains expectations about the current plan outcome, and each unexpected situation can be recorded as an exception. An explanation for this surprising event is then generated, and the plan is revised. At first expert knowledge was needed in the form of goals, strategies and computing expectations, but later work added the possibility of learning all these from demonstration [243].

The need for expert domain knowledge was reduced by using CBR and RL to create a learning goal-driven system [111]. This system improves its goals and knowledge over time, and can adapt to changes better than a non-learning agent. However, in practice, it did not out-perform the non-learning variant.

Hierarchical Planning Planning in RTS is a complex problem, and hierarchical systems aim to reduce this complexity using decomposition at different abstraction levels. For example, hierarchical task networks (HTNs) decompose goals into a series of sub-goals, at a lower abstraction level. The process continues in the same fashion until eventually the goals can be decomposed into concrete actions. While this approach can reduce the computational effort required for planning, HTNs generally need significant effort for encoding strategies and goal decompositions.

A hand-crafted HTN is used to play the RTS game Spring, and is able to defeat the built in scripted AI [129]. Learning the HTNs would be the next step in this research area, however this has only been tried in very simple domains and never in RTS games [158].

While these are the most popular methods for planning in RTS games, other approaches have been explored. For example, several scripted players are compared to a minimax and a Nash equilibrium player which are found to perform better than the scripted players [180]. Monte Carlo was also used to select high level strategies and showed promising results [37]. However, both these approaches use very simple RTS games.

While most research at the strategic level of RTS games is dedicated to planning, there are a few difficult planning sub-problems that are usually ad-

dressed independently. For example, two such sub-problems are choosing a good army composition and finding an optimal build order for the player's structures.

Army Composition can be part of a predetermined plan, or decided on-the-fly, taking into account the predicted enemy army composition. Choosing the army composition is a difficult subproblem by itself, as all unit types in an RTS game such as StarCraft have different attributes (walking or flying, attack damage, range and type, armor and health, size and speed). A CBR approach is used to this extent and the authors claim good results but do not present any quantitative evaluation [33]. There is very little work on selecting army compositions, researchers preferring selecting a specific plan instead and using a few scripted rules to slightly alter the army composition.

Build Order To reach a set goal of units and buildings, a player needs to construct different buildings according to their pre-requisites; the timing or order in which these buildings are constructed is called a build order. The problem of planning a build order can be described as a constraint resource allocation problem with concurrent actions [125]. A time intensive, off-line build order planner that uses evolutionary algorithms discovered a few build orders that proved very effective when implemented by humans [24]. Finally, multiple lower bound heuristics and abstractions (such as macro actions and income abstractions) were used to reduce the build order search effort and to produce close to optimal plans in real-time [40].

2.4 Holistic Approaches

As mentioned above, one way to tackle the large search space in StarCraft is by decomposing the problem of playing a RTS game into a set of smaller independent problems. These are usually solved by separate AI modules.

Holistic approaches propose a unified perspective and try to address the whole problem within a single framework (e.g., game-tree search). Until recently, there have been few attempts to implement such a system, notably ALisp [143] and Darmok [159], which combines case-based reasoning and learning from demonstration.

The research community is starting to focus again on this area, first by tackling the problem of global search in smaller scale RTS games such as μRTS [160, 161]. More recent research uses abstraction to only search over a high-level representation of the game state [232, 233]. The authors decompose the map into connected regions, and subsequently group units into squads taking their type into account. The space search is reduced because moves are then restricted to squads (which can go to a connected region, attack, or

defend). However, this approach deals mainly with combat and disregards some aspects of playing the full StarCraft game.

Adversarial Hierarchical Task Network (AHTN) planning combines game tree search with HTN planning [164]. The authors implement five different HTNs, at different abstraction levels. These HTNs produce game trees ranging from one identical to the tree that would be generated by minimax search applied to raw low-level actions, to a game tree with only one max/min choice layer at the top, followed by scripted action sequences. Experiments are conducted in μRTS, against several scripted and search based agents. The three HTNs with more abstract tasks displayed good performance. The most abstract one showed promise to scale well to more complex scenarios or games.

2.5 Deep Reinforcement Learning in RTS Games

In machine learning, traditionally there have been three approaches to learning: supervised, unsupervised and reinforcement learning. All three can be used to train deep neural networks to play games, in addition to other methods such as stochastic optimization approaches (e.g., neuroevolution) or hybrid methods that combine several of these approaches. In this section we briefly describe how these methods work and can be applied to RTS games, then follow up with a more in-depth explanation of RL concepts in Section 2.5.1 and finish by giving a historical overview of deep RL research applied to games, in particular RTS games.

Supervised Learning is a process through which an algorithm learns from a training dataset for which the correct answers are known. The algorithm iteratively makes predictions such as a label in classification problems or a real value in regression problems, and the difference between the given answer and the ground truth is used to update the model. Learning stops when an acceptable level of performance is achieved on the training data, with the goal being to generalize and achieve good results on new and unseen data.

Supervised learning requires a large amount of training examples, which often require human effort to be produced or labelled. Game replays can be used to learn which actions a human would perform in a given state [22]. Strategic choices performed by humans in StarCraft macro-management tasks were learned from game logs, and promising results were obtained when integrating the neural network in an existing bot [117]. For AlphaGo [195], 30 million Go positions from expert human games were used to train the algorithm in the first stage, followed up by self play and reinforcement learning in the second stage. If, however, there is not enough human data one could generate training data using other approaches. For example, in the research described in Chapter 7 we trained a CNN as a state evaluator using bots to play games

and label them as victories or defeats. Moreover, if there is an algorithm with good performance that is too slow to run in real time, supervised learning can be used to train a neural network to predict its actions, as shown in work described in Section 8.3.

Unupervised Learning techniques model the underlying structure or distribution in the data without access to any corresponding output labels or values. Patterns in the data are learned and used to cluster similar datapoints into groups, to discover rules that describe large portions of the data, or to reduce data dimensionality and compress it while maintaining its essential structure.

One popular unsupervised learning approach used in conjunction with deep networks is the *autoencoder*, which is a hourglass shaped neural network that learns to replicate its input. The first part of the network is called the encoder, which compresses the input data into a short code, while the second part – the decoder – decompresses the code into a copy of the original input. There is no need for labels, as the performance is evaluated by how well the reconstruction matches the original data. The goal is to learn a meaningful, shorter representation of the data. Autoencoders were used to learn a compressed state representation, and reinforcement learning methods were then applied in the reduced state space at a substantially lower computational cost [5]. 8×8 μRTS maps with information such as positions and health of all units were compressed into a representation (code) that uses only 15 neurons.

Other Approaches to training deep neural networks includes neuroevolution (NE), which besides learning the network's weights can be used to optimize its topology as well. These methods use stochastic noise to explore in the parameters space directly, and do not require computing gradients. Not relying on differentiability allows evolutionary algorithms to be used in conjunction with a wide range of algorithms, including search methods. NE methods have been proved a scalable alternative to recent RL approaches [181]. More details about applying NE methods to games can be found int [175].

A successful hybrid approach is AlphaGo [195], which combined supervised learning from human replays, reinforcement learning and Monte-Carlo tree search to defeat the world Go champion. Hybrid approaches have been applied in RTS games as well, for example to inform tree search [162] or to reduce the high dimensional input before applying a different method to learn how to play [5]. A similar approach to the latter used autoencoders for compressing but NE instead of RL for learning how to play a 3D shooter game [4], but has not been tested yet in RTS games.

2.5.1 Reinforcement Learning Approaches

In reinforcement learning, agents interact with an environment from which they receive a reward signal and their goal is to learn a behavior that maximizes the accumulated rewards. Games are easy to be modeled as environments for RL, with agents having to learn which of the available actions to take at each step to solve a Super Mario level or defeat a StarCraft enemy player. The reward $R(s)$ received for reaching a state s is usually close to zero for most states, possibly reflecting small changes in score between sequential states, and meaningful values are often sparse (e.g., large bonuses/penalties for winning or losing a game). Assigning credit to the many actions taken before receiving a reward signal is challenging, and below we introduce some of the main algorithms in RL designed for this problem.

One of the most popular approach is temporal difference (TD) learning, based on the idea that the utility value for a state is equal to the reward obtained at the current step for reaching that state plus the utility value of the next state [212]. This is implemented using the following update equation:

$$U(s) = U(s) + \alpha(R(s) + \gamma U(s') - U(s)),$$

where γ is the discount factor for utility of future states and α is the algorithm's learning rate. It will take one step for $R(s_t)$ to influence $U(s_t)$, and another step to influence the utility of a previous state $U(s_{t-1})$. With this update rule, the rewards' effect will propagate backwards in time until convergence.

This method doesn't take into account which action was chosen to reach the next state. There are other implementations of TD that learn utility values for state-action pairs instead. These methods are model-free, in the sense that agents can choose which action to take given a state without a state transition model. *Q-learning* is a popular model-free approach that computes utility values for states by taking the maximum $Q(s, a)$ value over the actions available in that state [238]. The update rule is:

$$Q(s, a) = Q(s, a) + \alpha(R(s) + \gamma \text{max}_a Q(s', a') - Q(s, a)).$$

A very similar algorithm exists, called State-Action-Reward-State-Action (SARSA) which updates estimates based on the a' action taken by the agent instead of the maximum estimate of possible next actions [178]:

$$Q(s, a) = Q(s, a) + \alpha(R(s) + \gamma Q(s', a') - Q(s, a)).$$

An agent's policy $\pi(s)$ is a function that takes the current environment state and returns which action to take. SARSA is an *on-policy* algorithm, because it uses the current policy's action a' to update Q-values, estimating the return assuming the current policy continues to be followed. In contrast, Q-learning is an *off-policy* algorithm, greedily choosing a' for updates. It assumes a greedy policy is followed when estimating the return, despite the fact that

any behavior policy can be used. Usually behavior policies are constructed to balance exploration of new actions and exploitation of current knowledge. Early in training it is more important to explore and not get stuck in local optima while later, when Q-values are more accurate, actions can be chosen more greedily.

Policy search is another approach in RL, and performs optimization directly in policy space. If $\pi_\theta(s, a)$ encodes the probability that action a is chosen in state s, then policy search optimizes the parameters θ via techniques like gradient descent. For example, the REINFORCE algorithm [246] uses the gradient $\nabla_\theta \sum_a \pi_\theta(s, a) R(s)$ to update θ, where $R(s)$ is the discounted cumulative reward received from s to the end of the episode. The result is a stochastic policy that samples actions and those that happen to eventually lead to good outcomes get encouraged in the future, while actions taken that lead to bad outcomes get discouraged.

Generally, value methods such as SARSA and Q-learning work well when there is a finite set of actions, while policy search is useful when the action space is continuous or stochastic. One issue with policy gradient methods is that all actions from a trace are updated either positively or negatively, depending only on the total reward at the end of the episode. Consequently, many samples are required until convergence. Actor-Critic methods address this issue by combining both value and policy estimation methods, and making updates at each step instead of waiting until the end of the episode as in REINFORCE [213]. A critic model learns a value function using TD learning, which is used to evaluate the current policy. A second model – called the actor – uses this value to replace $R(s)$ from the policy gradient update rule to learn and improve the current policy.

2.5.2 Overview of Deep RL Methods used in RTS Games

This subsection presents some of the more popular methods used in deep RL. Many of these algorithms were developed and applied first on Atari games, and then extended to more complex domains like RTS games. Therefore, we first introduce the methods within their original application context, and then mention relevant work that uses or extends this research to RTS games.

Deep Q-Networks (DQN) outperformed approaches such as NE [95] and SARSA [18] and was the first algorithm that learned expert-level control in Atari games, surpassing human expert scores in three games. A neural network with two convolutional and one fully connected layer was used to approximate Q-values, and was later extended by adding a second copy of this network. This target network generates Q-values that are used to compute the loss during training and is updated more slowly compared to the primary network, increasing stability and preventing value estimations to spiral out of control. Human scores were surpassed in 29 of the 49 Atari games [150]. DQN is a

sample efficient algorithm, using experience replay to store past experiences (tuples containing a state, action chosen, reward received and next state) in a replay memory. Batches of random – and hopefully uncorrelated – experiences are drawn from the memory and used for updates, forcing the network to generalize beyond what is immediately doing in the environment.

There are many extensions to DQN algorithms, a few being:

- *Double DQN* used double Q-learning [93] to decorrelate the selection of the best action from the evaluation of that action, reducing overestimation of Q-values [94].

- *Prioritized experience replay* replaces the uniform sampling of experiences from the memory replay with an approach that samples significant – based on the transition error – transitions more frequently, and outperforms the original version in 41 of the 49 Atari games [187].

- *Dueling DQN* uses a network architecture that, after the convolutional layers, is split into two separate representations: the value function $V(s)$ and advantage function $A(s, a)$ [237]. They are combined together at the final layer to obtain Q-values: $Q(s, a) = V(s) + A(s, a)$. Learning how valuable states are is thus decoupled from learning the effect of every action for each state, resulting in more robust estimates.

- *Distributional DQN* learns to approximate the return value distribution instead of estimating a single value as output, providing a richer set of predictions and a more stable learning target [17]. This extension, without further algorithmic additions, surpassed gains made by double DQN, the dueling architecture and prioritized replay.

- *Noisy Nets* perturbs the network weights representing the agent's policy to improve exploration efficiency [65]. The amount of noise added to each parameter is learned during training.

- *Rainbow* integrates the above five DQN extensions and multi-step learning into one model, proving that these are largely complementary and their combination led to new state-of-the-art results [105].

- *Deep Recurrent Q-Learning (DRQN)* provides 'memory' functionality by replacing the fully connected layer(s) with one or two recurrent layers containing GRU or LSTM cells [96]. It helps in partial observability settings, especially in games such as first person-shooters [131].

- There are various *distributed* versions of DQN, using multiple agents to collect experiences into a shared memory replay. For example, Gorila [157] obtained higher scores than its non-distributed counterpart in 41 of the 49 Atari games.

While all the above approaches were applied to complete Atari games, RTS games are more complex environments and most research focuses on either sub-problems or restricted game versions. In addition, the lack of the in-game scoring present in console games makes rewards very sparse (e.g., winning or losing a game) and *reward shaping* [52, 57] based on the difference between statistics such as damage dealt and damage incurred is generally used to reduce this delay. Deep learning methods applied in RTS games have thus focused mostly on controlling units in combat scenarios, where feedback on actions taken is obtained more easily and problem complexity can be increased via the number of controlled units.

A DQN baseline was implemented for StarCraft combat scenarios, and compared to a greedy approach that sequentially choses actions for agents and searches in policy space using zero-order gradient optimization [235]. This latter method – GreedyMDP with Episodic Zero-Order Optimisation (GMEZO) – successfully learned non-trivial strategies for controlling armies of up to 15 units while basic DQN and REINFORCE approaches struggled.

Independent Q-Learning (IQL) [223] is often used to decompose the multi-agent problem into simpler, individual agent learning tasks by treating all other agents as part of the environment. However, IQL is not stable when used in conjunction with methods such as experience replay, as the multiple learning agents make experiences lose relevance more quickly. This issue was addressed and DQN with prioritized experience replay and a few stability fixes has shown improved performance in small StarCraft combat scenarios with up to 5 units on each side [63].

A different way to address multi-agent problems is to enable differentiable communication between the agents, which allows them to cooperate. In CommNet, each agent is controlled by a deep network which has access to an all-to-all communication channel through which a summed transmission of other agents is broadcasted [209]. This approach outperformed both independent learning and fully connected models on simple combat tasks of up to 10 units per side.

Actor-Critic Methods are built with two components: the actor which maintains and updates the agent's policy and the critic which is used for state evaluation. In Advantage Actor-Critic (A2C), the critic uses an advantage function rather than a value function, and is often defined as a synchronous version of the Asynchronous Advantage Actor-Critic (A3C) [148]. In A2C and A3C multiple agents interact with the environment independently and simultaneously, and the diverse experiences are used to update a global, shared agent network. They are equivalent mathematically, with the practical implementation difference that A3C agents send updates and retrieve parameters to/from the central server asynchronously while A2C uses a deterministic implementation that waits for each actor to finish its segment of experience and batches them together. A2C and A3C have been found to perform similarly,

with A2C utilizing GPUs more efficiently while A3C is preferred on CPU-only architectures [250].

The main advantage of A2C and A3C methods is their capacity for mass parallelization, which reduces training time in a nearly linear fashion with the number of parallel executing agents [148]. The agents' uncorrelated experiences serve a similar function to the experience replay from the off-policy DQN-based methods and improve learning stability. Compared to DQN however, actor-critic methods are less sample efficient.

Trust Region Policy Optimization (TRPO) is an off-policy gradient method that uses an actor-critic architecture, but modifies how the actor updates the policy parameters [190]. It constraints the size of the policy updates and avoids parameter updates that change it too much at one step, which results in improved training stability. This is done by computing the KL divergence between the old and the new policy is relatively complicated, which is relatively complicated. The newer Proximal Policy Optimization (PPO) method greatly simplifies this computation using a clipping function while retaining similar performance [191].

A3C was used as a benchmark in the StarCraft II Learning Environment (SC2LE), in work that advocated using StarCraft II as a new and challenging environment for exploring deep RL algorithms and architectures [236]. Three different network architectures were compared and agents comparable to novice players were trained for several minigames. However, no significant progress was obtained on the full game, the basic A3C implementation failing to defeat even the easiest difficulty (level 1 out of 10) built-in AI.

Counterfactual Multi-Agent Policy Gradients (COMA) is an actor-critic method that uses a counterfactual baseline to marginalise out a single agent's action, while keeping the other agents' actions fixed [62]. It outperformed independent actor-critic methods, IQL, and GMEZO in StarCraft combat scenarios with up to 5 units per side.

Multiagent Bidirectionally-Coordinated Network (BiCNet) is a vectorized version of actor-critic that uses bidirectional neural networks to exchange information between agents [171]. In 4 out of 5 scenarios with 5 up to 20 units per side, BiCNet outperformed other baselines such as an independent controller, CommNet and GMEZO.

A PPO algorithm, extensive feature engineering and five networks with a single LSTM layer each were used to produce competitive agents for the full 5 vs. 5 game of Dota 2, with only minor item and rule restrictions [168]. Some of these restrictions were later lifted and the AI defeated several amateur teams but lost against professional players [167].

Finally, an actor-critic framework used in conjunction with relational RL and a more data efficient distributed algorithm [60] achieved state-of-the-art results in six SC2LE minigames (movement and combat based scenarios that are simpler than the full game) and its performance surpassed that of the human grandmaster in four of them [254].

Other Approaches Credit assignment from team reward signals is challenging, especially as the number of agents increases. One approach to deal with this issue is to transform multiple agent interactions into interactions between two entities: a single agent and a distribution of the other agents [252]. This technique improved agents trained with either IQL or A2C over their original versions on combat scenarios with 64 units per side. Another approach to solve credit assignment is to use a *value decomposition* network (VDN) to learn how to represent the total Q-value as sum of per-agent, individual Q-values [211]. This method was extended in QMIX [174] by replacing the sum operation with a mixing network that allows non-linear combinations of Q-values. QMIX outperformed IQL and VDN in small SC2LE scenarios of up to 8 units per side.

Finally, recent research published just before the submission of this thesis by two different groups simultaneously uses *hierarchial RL* to speed up learning and to explore the environment more efficiently. These are the first instances of learning bots that play full scale and non-reduced versions of RTS games on a comparative level to good human players. Firstly, a two-level hierarchial architecture is constructed, with policies running at two different timescales [170]. Abstraction is used to reduce the action space, the low level actions consisting of macro-actions automatically extracted from experts' trajectories. The PPO algorithm is used for training the network, which achieves 93% win rate against level 7 built-in SC2LE AI on the full StarCraft II game played on a small 64×64 map. The second group adopts a set of macro actions as well, this time designed using hard coded prior knowledge [210]. While similar to previous research that uses hand coded macro actions in a customized mini RTS game [228], the authors implement a much larger set of rules – 165 macro actions. On top there is a single controller trained with dueling double DQN or PPO. On a SC2LE tournament map of 152×136 tiles, both versions defeat level 1 to level 9 AI in more than 90% of the games. Against level 10, the DQN version achieves 71% win rate and PPO 81%.

Chapter 3

Predicting Opponent's Production in Real-Time Strategy Games with Answer Set Programming

Abstract

The adversarial character of real-time strategy (RTS) games is one of the main sources of uncertainty within this domain. Since players lack exact knowledge about their opponent's actions, they need a reasonable representation of alternative possibilities and their likelihood. In this article we propose a method of predicting the most probable combination of units produced by the opponent during a certain time period. We employ a logic programming paradigm called Answer Set Programming, since its semantics is well suited for reasoning with uncertainty and incomplete knowledge. In contrast with typical, purely probabilistic approaches, the presented method takes into account the background knowledge about the game and only considers the combinations that are consistent with the game mechanics and with the player's partial observations. Experiments, conducted during different phases of StarCraft: Brood War and Warcraft III: The Frozen Throne games, show that the prediction accuracy for time intervals of 1-3 minutes seems to be surprisingly high, making the method useful in practice. Prediction error grows only slowly with increasing prediction intervals – almost in a linear fashion.

3.1 Introduction

Real-time Strategy (RTS) games represent a genre of video games in which players must manage economic tasks like gathering resources or building new bases, increase their military power by researching new technologies and training units, and lead them into battle against their opponent(s). RTS games serve as an interesting domain for Artificial Intelligence (AI) research, since they represent well-defined complex adversarial systems [28] which pose a number of interesting AI challenges in the areas of planning, learning, spatial/temporal reasoning, domain knowledge exploitation, task decomposition and – most relevant to the scope of this article – dealing with uncertainty [166].

One of the main sources of uncertainty in RTS games is their adversarial nature. Since players do not possess an exact knowledge about the actions that their opponent will execute, they need to build a reasonable representation of possible alternatives and their likelihood. This work specifically tackles the problem of predicting opponent's unit production. The method presented here allows artificial players (bots) playing RTS games to estimate the number of different types of units produced by their opponents over a specified time period.

Various problems related to uncertainty and opponent behavior prediction in RTS games have already been addressed in recent years (an extensive overview can be found in [166]). For example, Weber and Mateas [239] proposed a data mining approach to strategy prediction, Dereszynski et al. [51] used Hidden Markov Models to learn the transition probabilities within building construction sequences, Synnaeve and Bessiére [216] presented a Bayesian semi-supervised model for predicting game openings (early game strategies). Weber et al. [242] used a particle model to track opponent's units out of vision range and hidden semi-Markov models were used in conjunction with particle filters to predict opponent positions in [106]. Kabanza et al. [118] used HTNs to recognize opponent's intentions, while extending the probabilistic hostile agent task tracker (PHATT) by [73]. We supplement this work by introducing a logic-based solution to yet another subproblem from this area – predicting opponent's unit production. This particular challenge is relevant to any conventional RTS game and takes almost the same form in all games of this genre. Typically, the differences are only in the input data: costs and production times of individual unit types and their technological preconditions. Reasoning behind the prediction task itself is, however, not dependent on game-specific information.

Expert knowledge about complex domains, such as RTS games, is often extensive and hard-coding the reasoning for a number of different games in imperative programming languages tends to be quite inconvenient and time-consuming. Therefore it makes sense to consider using *declarative knowledge representation paradigms*, many of which are well-suited for this kind of problems.

Compared to traditional imperative methods, a declarative approach allows for easier expression of typical RTS mechanics. Game-specific knowledge can be separated from common reasoning mechanisms by employing modular design patterns. A specific paradigm of logic programming called Answer Set Programming (ASP) [74, 75, 8] was selected for the solution presented in this work. Its nonmonotonic semantics with negation as failure is well suited for reasoning with uncertainty and incomplete knowledge, and has already been used for prediction tasks outside the computer games domain [55, 71]. In contrast with purely probabilistic approaches, using ASP to predict opponent's unit production allows for the exploitation of background knowledge about the game, so that only those unit combinations that are reliably consistent with game's rules and player's partial observations (both encoded by ASP) are considered.

3.2 Answer Set Programming

Answer Set Programming has lately become a popular declarative problem solving paradigm with a growing number of applications. In the area of game AI, ASP has been used in attempts to implement a "general player", either in combination with other techniques [153], or in ASP-only projects like the one presented in [1]. Additional work has been done on translating general Game Description Language (GDL) into ASP [227]. However, to the best of our knowledge, the only published application of ASP specifically for RTS gameplay dealt with wall-in building placement in StarCraft [34].

The original language associated with ASP allows the formalization of various kinds of common sense knowledge and reasoning, including constraint satisfaction and optimization. The language is a product of a research aimed at defining a formal semantics for logic programs with default negation [74], and was extended to allow also a classical (or *explicit*) negation [75] in 1991. An ASP *logic program* is a set of *rules* of the following form:

$$h \leftarrow l_1, \ldots, l_m,\ not\ l_{m+1}, \ldots\ not\ l_n.$$

where h and $l_1, \ldots, l_n$ are classical first-order-logic literals and *not* denotes a default negation. Informally, such rule means that "if you believe $l_1, \ldots, l_m$, and have no reason to believe any of $l_{m+1}, \ldots, l_n$, then you must believe h". The part to the right of the "$\leftarrow$" symbol ($l_1, \ldots, l_m,\ not\ l_{m+1}, \ldots\ not\ l_n$) is called the *body*, while the part to the left of it (h) is called the *head* of the rule. Rules with an empty body are called *facts*. Rules with empty head are called *constraints*. We also say that a literal is *grounded* if it is variable-free. A logic program is called *grounded* if it contains only grounded literals.

The ASP semantics is built around the concept of *answer sets*. Consider a grounded logic program Π and a consistent set of classical grounded literals S. We can then get a subprogram called *program reduct of* Π *w.r.t. set* S (denoted Π^S) by removing each rule that contains *not* l, such that $l \in S$, in

its body, and by removing every "*not l*" statement, such that $l \notin S$. We say that a set of classical grounded literals S is *closed* under a rule a, if holds $body(a) \subseteq S \Rightarrow head(a) \in S$.

Definition 1 (Answer Set) *Let* Π *be a grounded logic program. Any minimal set* S *of grounded literals* closed *under all the rules of* Π^S *is called an* answer set *of* Π.

Intuitively, an answer set represents one possible meaning of knowledge encoded by a logic program Π and a set of classical literals S. In typical case, one answer set *corresponds to one valid solution of a problem* encoded by our logic program.

In this work, returned answer sets will correspond to different combinations of units that might be trained by the opponent during the following time period, which are consistent with the game rules and current partial observations.

3.3 Predicting Unit Production

This work assumes that modeled game mechanics are consistent with the following conventions, typical for RTS games:

- Units are trained in production facilities – specific types of structures such as Barracks, Workshop, Gateway, etc.
- Producing units or structures requires resources and time.
- There is a continuous income of consumable resources (e.g., minerals, gold) that can be approximated based on a number of certain units (referred to as workers).
- Renewable resources (e.g., supply, accommodation or power) are obtained by constructing certain structures.
- Certain unit types may have specific technological prerequisites (availability of certain structures or technologies).

Good examples of games that satisfy all of these conditions can be found in the *Age of Empires*, *WarCraft* or *Command & Conquer* franchises. Implementations presented in this article use two such games: *StarCraft: Brood War* and *Warcraft III: The Frozen Throne*, both developed by Blizzard Entertainment.

Since the goal is to predict the most probable among the valid combinations of units, the following need to be accomplished:

- generating all the unit combinations that could be trained, given the opponent's observed production facilities,

- removing the unfeasible combinations (units take time to train, and they cost resources and supply),
- selecting the most probable valid combination, using the information from recorded games.

The game was accessed in real time via BWAPI[1] in case of StarCraft and by custom replay parser in case of WarCraft III. At each relevant time step, a logic program representing the partial knowledge about current game situation was constructed and the answer sets of this program were computed using the ASP solver called CLASP [72], which supports several convenience features like generator rules, aggregates or optimization statements. When reading the code fragments in this section, one should note that the parameters starting with uppercase letters are variables (e.g., "`T`", "`Z`", "`Number`"), while the others are constants corresponding to objects from the domain or numbers (e.g., "`gateway`" or "`40`"). The following examples focus on the StarCraft game, and therefore StarCraft-specific terminology is used, but only for the names of units and structures; the proposed method can be easily converted for other RTS games.

3.3.1 Generating Unit Combinations

First of all, the maximum number of each type of unit that can be trained using the opponent's production facilities needs to be computed. For example, a Gateway can be used to train Zealots (each takes 40 seconds) or Dragoons (50 seconds). For the sake of simplicity, only these two types of units are considered in the examples below. Over a time interval of 200 seconds, the maximum number of Zealots that can be trained will be 5, and the maximum number of Dragoons 4. If the opponent had two Gateways instead, then these numbers would double, to 10 and 8 respectively. For example, the code below computes the maximum number of Zealots:

```
timeGateway(Number*T) :- built(gateway,Number), interval(T).
maxZealots(Z)         :- timeGateway(T), Z = @min(T/40,24).
```

Computation time generally grows exponentially with the maximum number of allowed units, so certain limits were imposed (in the above case, *maxZealots* was limited to 24). Note that this maximum number represents how many units can be trained independently of other units, so obviously it is not possible to have 4 Zealots and 3 Dragoons within 200 seconds, using a single Gateway. All unit combinations can be generated using the following code – however, many of them will not be feasible:

```
1 {zealots(0..N)}  1    :- maxZealots(N) , N > 0.
1 {dragoons(0..N)} 1    :- maxDragoons(N) , N > 0.
gateway_units(Z,D):- zealots(Z), dragoons(D).
```

[1] `http://github.com/bwapi/`

3.3.2 Removing Invalid Combinations

Next, the combinations that exceed the time limit need to be removed using the following constraint:

```
:- gateway_units(Z,D), timeGateway(T1), (Z*40 + D*50) >= T1.
```

Assuming that the opponent tries to spend everything as soon as possible, the maximum quantity of resources available over the next time period can be accurately approximated using the number of current workers. The *resources per second* gathering rate for a worker is a well known constant. We assume that the opponent continues producing workers at a steady rate, which is what skilled players usually do in StarCraft.

Knowing the maximum amount of resources, more constraints can be added to prune out all the combinations that are too expensive to train. However, there is one more restriction to be imposed, concerning the *units supply*. For example, all the Gateway units take up 2 supply each, and a player cannot exceed the current population limit. However, by building Pylons (specific structures that can be built for 100 minerals), a player raises the supply cap by 8 – and 4 extra Zealots can be trained, for example. If a combination uses more supply than is currently available to the player, building as many Pylons as necessary for sustaining it is also imposed.

3.3.3 Choosing the Most Probable Combination

The previous subsections have shown how to produce many possible unit combinations – answer sets representing different valid armies which can be trained over the selected time interval. However, the goal is to be able to accurately *predict the most probable* such *combination*. Consequently, only one answer set should be selected from all the possibilities. One reasonable approach is to assume that the opponent tries to spend the maximum amount of resources or, equivalently, to have as few remaining resources as possible. This can be easily accomplished using CLASP's *#minimize* statement.

Alternatively, it might be useful to predict the most dangerous valid unit combinations, so the player can prepare against them in particular. However, this requires quantifying the strength of a group of units relative to other units or states which is a hard problem, out of scope of this article (simulations [43] or machine learning [206] might help).

Finally, the approach chosen in this paper is to actually pick the *most probable* unit combination for a given game phase, based on a set of recorded games – replays [2]. Thus, the goal is to output the army combination most

[2]Note that these replays should be selected carefully. For example, if the probabilities were computed using mostly professional human vs. human games, the prediction accuracy against beginners or bots might suffer. We worked with the replays of high-level human vs. human StarCraft games from [219] and a collection of replay packs from different WarCraft III tournaments

likely to be seen from the opponent, or at least sort the answer sets according to such a criterion. The first step is to compute the probability of the opponent having Z zealots, D dragoons, and so on, at a specific $Time$, denoted $P(Z, D, \cdots, Time)$. Then, the answer sets are sorted according to this value. An obvious simplification considered was to assume that training a unit is *independent* of training the others. With this simplification, one can express the probability of the whole unit combination at a given time as:

$$P(Z, D, \cdots, Time) = P(Z, Time) \cdot P(D, Time) \cdot \ldots$$

Finding the probability of our opponent having Z zealots, etc. at a specific time is an easy task, accomplished by parsing a large set of replays and computing it as follows:

$$P(Z, T) = \frac{\text{Nr. of replays with exactly Z zealots trained at time T}}{\text{Total nr. of replays}}.$$

Even thought this simplifying assumption is quite strong, as there are considerable correlations between training certain unit types, it will be shown that it can be used for prediction with sufficient accuracy in the next section. Once able to compute the probability of specific unit combination, CLASP's optimization statements are used to enforce the selection policies discussed above:

```
#maximize [p(X) = X @ 2].
#minimize [remainingRes(M,G) = M+G @ 1 ].
```

The implementation used for the experiments combines two selection policies. First and foremost, it maximizes the likelihood of the combination (priority 2). If there are two equally probable answer sets, the one that leaves the opponent with less remaining resources (M – minerals and G – gas) is favoured (priority 1).

3.4 Experiments and Results

This section describes the conducted experiments and ex- plains the output of the implemented program. Afterwards, the metrics used for quantifying the prediction accuracy are introduced, and the obtained results are presented and discussed.

To the best of our knowledge, there have been no other published solutions to the problem of opponent's unit production, and so it is impossible to empirically compare the performance of the implemented method to alternatives.

3.4.1 Experiment Design

The experiments were conducted in four different game times: at 5^{th} ,10^{th} ,20^{th} and 30^{th} minute after the start of the game for StarCraft and at 5^{th} ,10^{th} ,15^{th} and 20^{th} minute for WarCraft III. The first time is usually associated with

the early-game phase, next two with mid-game and the last one is usually classified as a late-game phase. For each experiment, the units the opponent will have 1, 2 and 3 minutes in the future are predicted. 300 StarCraft and 255 WarCraft III replays were used for the evaluation. However, some games took less than 30, 20, or even 10 minutes to finish. For example, in the last experiment (30^{th} minute), only 65 out of the initial 300 StarCraft replays were still valid, and only 34 WarCraft III replays were valid after 20 minutes (a replay is considered valid if at least one unit is trained after the end of the prediction time).

Below is a sample output of the program, representing a three-minute prediction from 10^{th} of a StarCraft game. It took less than 60ms to compute, and it returned four answer sets. Below the prediction, there are the actual units trained by the player in this interval – 7 Dragoons (second unit type trained in the Gateway). Note that it is needed to keep the optimization scores and probabilities unnormalized, because CLASP can only work with integers. For instance, answer set 1 below has $prob = 5312104$ which corresponds to a probability of $p = 0.00053$. The optimization score the program is trying to minimize is just a constant minus the unnormalized probability.

```
------- Predicting for player 0 time 9000 to 11700 --------
        ( 10 mins, after 3 )

---------------------- CLASP output ----------------------
clasp version 2.1.1 Solving...
Answer: 1
gateway_units(4,4,0,0) robotic_units(0,0,0)
stargate_units(0,0,0,0) remainingRes(2202,866) prob(5312104)
Optimization: 112226583
Answer: 2
gateway_units(1,4,0,0) robotic_units(0,0,0)
stargate_units(0,0,0,0) remainingRes(2602,866) prob(5720728)
Optimization: 111817959
Answer: 3
gateway_units(0,7,0,0) robotic_units(0,0,0)
stargate_units(0,0,0,0) remainingRes(2227,716) prob(6803028)
Optimization: 110735659
 Answer: 4
 gateway_units(1,6,0,0) robotic_units(0,0,0)
 stargate_units(0,0,0,0) remainingRes(2252,766) prob(7266870)
Optimization: 110271817
OPTIMUM FOUND

Models      : 1
  Enumerated: 4
  Optimum   : yes
Optimization: 110271817
Time        : 0.053s
CPU Time    : 0.031s

---------------- Units trained in reality ----------------
  Gateway: 0 7 0 0
  Robotics: 0 0 0
  Stargate: 0 0 0 0
```

3.4.2 Evaluation Metrics

The number of units a player trains between two time stamps is predicted. There are 11 different types of units in the Protoss faction (4 from the Gateway and Stargate each, and 3 from the Robotics Bay), so the obtained answer set is converted to a 11-dimensional vector. To keep things easy to understand, the examples below only show the first 4 elements of this vector – the units produced from the Gateway. Assume that the real number of trained units is shown on the first line, and that the prediction for this situation is the second line:

Zealots	Dragoons	High Templars	Dark Templars	7 more
7	3	0	2	$\cdots$
5	3	0	3	$\cdots$

The difference between these two has to be computed in order to obtain an accuracy measure. The *root-mean-square error* (RMSE) was chosen, a very popular tool for this type of tasks. It is frequently used to measure differences between values predicted by a model or an estimator and the values actually observed:

$$\text{RMSE} = \sqrt{\frac{\sum_{i=1}^{n}(x_{1,i} - x_{2,i})^2}{n}}$$

where n is a number of values predicted, $x_{1,i}$ are the real values observed and $x_{2,i}$ are the predictions.

Applied to the previous example:

$$\text{RMSE}_{1:4} = \sqrt{\frac{(7-5)^2 + (3-3)^2 + (0-0)^2 + (2-3)^2}{4}} = 1.118$$

If the prediction is off by one unit for *every* unit type, then a resulting RMSE of 1 is obtained. But, as presented in the example, if the prediction is wrong by more units (two zealots), even if most of the others are exactly right, a RMSE bigger than 1 is obtained. This happens because RMSE penalizes wrong predictions more, the more wrong they are, by squaring the difference. With this in mind, a RMSE of approximately 1 for the accuracy on all eleven units is more than decent and can be caused for example:

- by being 1 unit wrong for all unit types,
- or at the extreme, being right for 10 unit types and wrong for the last one by 3.

In the example presented in the previous subsection, the output consisted of 4 answer sets in decreasing order w.r.t. the CLASP optimization score. Their RMSE values are 1.5075 (answer set 1), 0.9534 (answer set 2), 0 (answer set 3) and 0.4264 (answer set 4).

The last answer set printed (nr. 4) has the best optimization score, and is called the *optimal* answer set. Since lower optimization score does not necessarily mean better RMSE, the *best* answer set is also recorded – the one with the lowest RMSE (which in the example is answer set nr. 3). This is the closest one could get to the values actually observed, assuming a perfect job with the optimization statements and heuristics.

As a simple illustrative baseline to compare against, the trivial per-unit average (AVG) over all the valid replays was used, with some minor modifications. For example, for a target prediction the average across the replays is 5 Zealots and 2 Dragoons, for a total of $7 \times 2 = 14$ supply. However, if in the current game the player does not have the required structure for producing Dragoons, the AVG prediction is modified to 7 Zealots and 0 Dragoons instead, to maintain the average target supply of 14. Naturally, the proposed method is expected to achieve higher accuracy than this baseline, since it takes into account background knowledge and partial observations.

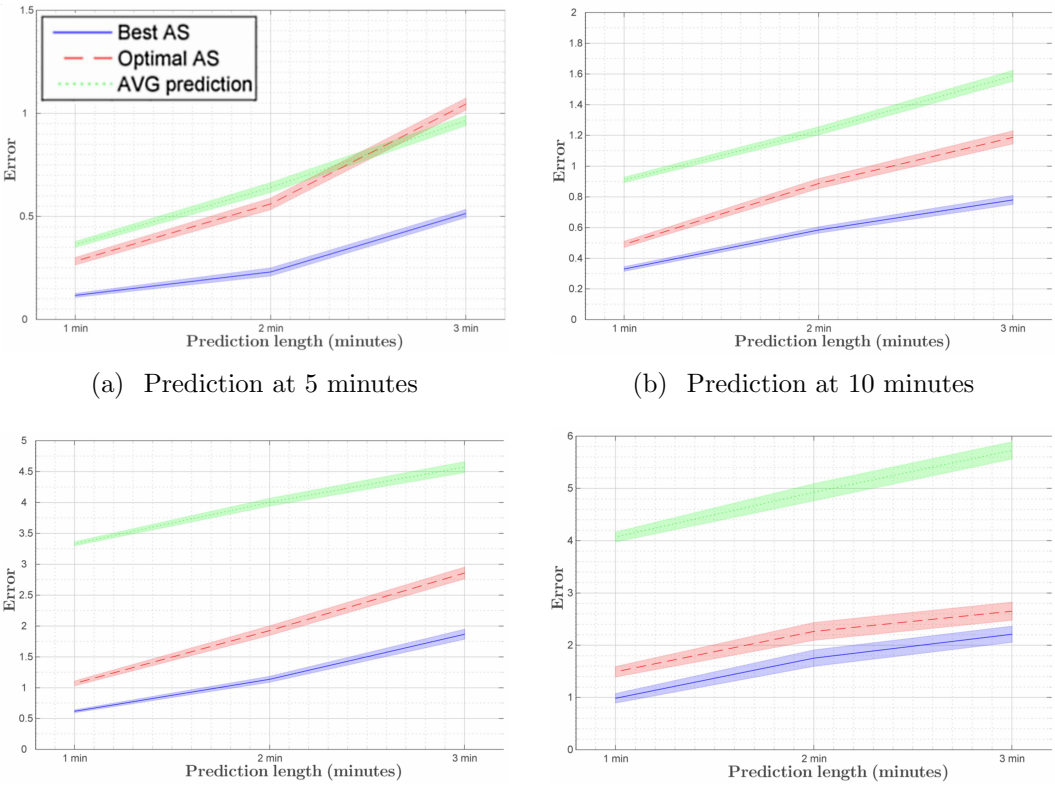
3.4.3 Results

The algorithm had been run for all 300 StarCraft and 250 WarCraft III replays, then the values for three quantities of interest were averaged: RMSE of AVG baseline, best and optimal answer sets. The results are shown in Figure 3.1 for StarCraft and Figure 3.2 for WarCraft III . Standard deviation divided by the square root of the number of examples is shown as the shaded quantity in the plots. This measure is known as *standard error*; one margin of standard error gives 68% confidence (of the result being within the shaded area).

As expected, the best answer set has a significantly lower error than the rest. However, the longer the game, the closer the optimal answer set gets to the best answer set. Also, the deviation is larger as the time increases, partly because there are more answer sets computed, and partly due to the fact that the replay training set is smaller.

Both optimal and best answer sets have much lower RMSE than the AVG baseline, except for the *early-game* prediction at 5^{th} minute. At this time, there are many instances in which no fighting units are trained, or they are produced just seconds after the prediction range, which makes the prediction more difficult. While it is likely that the correct answer set is among the ones printed by CLASP (best answer set still has significantly lower RMSE), the optimal answer set chosen is quite close to the baseline in this case. Biasing the algorithm to choose the unit combinations that maximize the amount of resources spent seems to be counter-productive in this phase.

The most important area is around 10^{th} to 20^{th} minute – the *mid-game*. Before that, the game is dominated by the build order prediction and is more strategy oriented. After 20 minutes, many matches are finished already or the winner is obvious, and players do not train units so much. The prediction error at 10^{th} minute (until 11^{th} ,12^{th} and 13^{th} min) is around 0.5, 0.9 and



(a) Prediction at 5 minutes

(b) Prediction at 10 minutes

(c) Prediction at 20 minutes

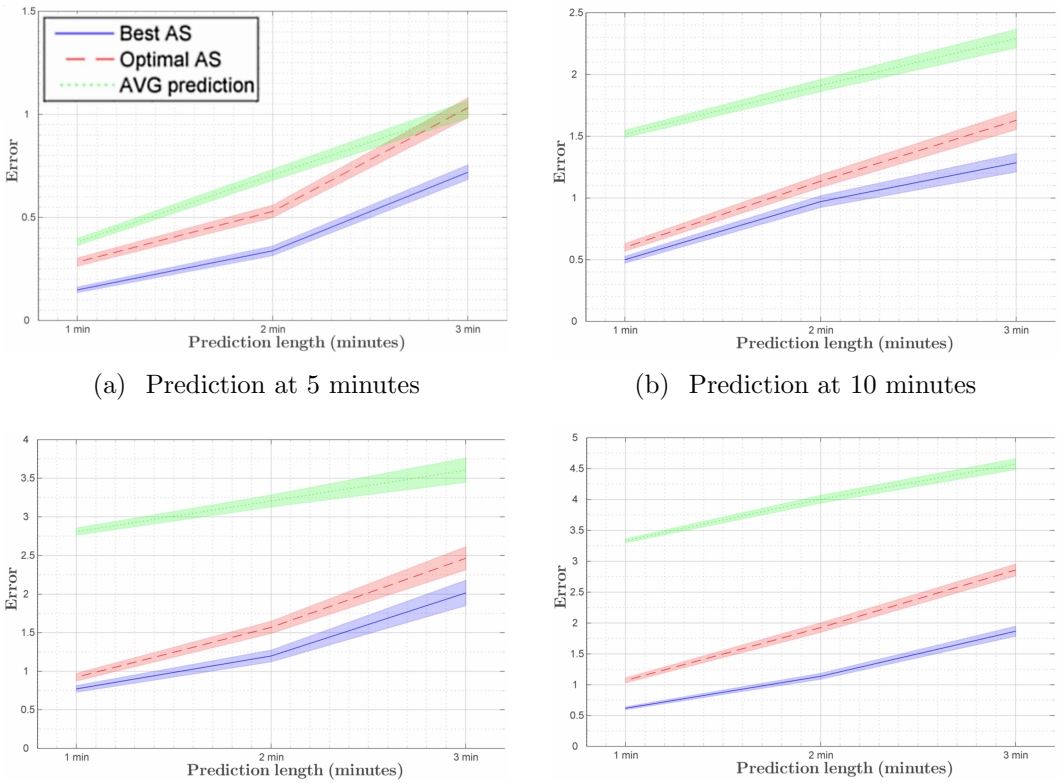
(d) Prediction at 30 minutes

Figure 3.1: Average predictions over StarCraft replays, shown as RMSE for best and optimal answer set, as well as a baseline that predicts the average number of units trained across all replays. The number of replays varies from 300 (5^{th} min) to 65 (30^{th} min).

1.2 for StarCraft and 0.6, 1.1 and 1.6 for WarCraft III, which is a satisfactory result. Predictions at this time tend to be more difficult in WarCraft III, where player's resource spending patterns temporarily deviate from typical RTS: they hire their second "hero" and purchase an equipment for him/her. However, the errors for the last two game times are very similar for StarCraft and WarCraft III, the RMSE being close to 2 for the 2 minute prediction and a little under 3 for the 3 minute predictions. This level of accuracy could certainly be useful in a real game and help the AI make better decisions towards what units to train to beat the adversary.

3.4.4 Running Times

For StarCraft, the running times of the prediction are on average around 10ms, 100ms, 2s and 8s for the four game phases (5^{th}, 10^{th}, 30^{th} and 30^{th} minute). This happens because after 30 minutes a player may have more than 10 pro-



(a) Prediction at 5 minutes

(b) Prediction at 10 minutes

(c) Prediction at 15 minutes

(d) Prediction at 20 minutes

Figure 3.2: Average predictions over WarCraft 3 replays, showing the same quantities as Figure 3.1. The number of replays varies from 255 (5^{th} min) to 34 (20^{th} min).

duction facilities, resulting in many valid combinations. Usually 20% of the time is spent either grounding or reading the grounded program, especially as the prediction time increases. A 10 second cut-off was imposed for the CLASP solver; we force the program to terminate after 10 seconds and return the best among the answer sets it managed to find so far.

The same prediction problem could in theory be solved using a "brute-force" approach with identical results in some kind of imperative programming language by (1) looping over all the unit combinations, (2) checking the validity of each combination, (3) assigning the probability to it and (4) selecting the most probable one. This in fact resembles the procedure performed by the CLASP ASP solver: (1) looping over the answer set candidates invoked by the generator rules, (2) eliminating the candidates that violate some of the constraints, (3) computing the optimization score and (4) selecting the best answer set according to it. Even though both of these problems have a high worst-case time complexity, ASP solvers like CLASP are quite fast in the average case, since they employ a variety of heuristic search strategies in the process, including *BerkMin*-like, *Siege*-like or *Chaff*-like decision heuristics [72] and can reduce the search space using the optimization statements.

3.5 Conclusion

A prediction system based on the Answer Set Programming paradigm was successfully built and its performance evaluated. As experiments suggest, it can accurately predict the number and type of units a StarCraft or WarCraft III player trains in a given amount of time. Very good results were demonstrated during mid-game and late-game (10^{th} to 20^{th} minute), for prediction length of 1 to 3 minutes. This is the interval in which the prediction of opponent's army production would be most useful.

The presented implementation is faster than expected from an ASP program. For the targeted interval described above, the running times averaged around 200ms, which is acceptable for a high-level decision making in RTS games AI.

It is clear from described results that building a simple probabilistic framework works with decent results, and that using ASP in strategy games brings certain advantages. A few other interesting tasks, suitable for future work, would be:

- Smoothing over the probabilities, or working with a Gaussian framework. We currently only consider $P(Zealots, Time)$, but looking from $Time - \Delta t$ to $Time + \Delta t$ and using weights, or trying to fit a Gaussian to the data might work better.

- Replacing probability optimization with a learned heuristic (combination of probabilities, resources spent, strength of units, and any other factors).

- Employing the *reactive* ASP paradigm [70], which can be viewed as an extension to ASP, and is ideal for dynamically changing domains like RTS games. Online ASP solvers like oClingo [3] process an initial logic program, compute the answer sets, but do not terminate. Instead they keep running as a server applications and wait for further knowledge updates (observations), while preserving and reusing all the previous computations. With this speedup, an artificial RTS player would potentially be able process new observations and generate predictions with much higher frequency.

3.6 Contributions Breakdown and Updates Since Publication

Most of the work in this chapter was performed by Marius Stanescu. Michal Čertickỳ gathered data and helped running experiments, as well as assisted in writing the published article.

[3] http://www.cs.uni-potsdam.de/oclingo/

To my knowledge, no research has been published since this chapter was written on predicting what size of type of army an opponent can train during a game. Build order prediction, or even more generally, strategy prediction is a popular related problem. However, it presents a different challenge: first one recognizes a player's strategy, then classifies it into a set of known or extracted strategies, and finally might switch his own strategy accordingly. This approach type assumes a set of strategies to choose from, and works on a high level without being able to provide accurate army estimations. Most build order methods are optimization algorithms that do not take into account or try to predict army sizes. They would indeed benefit from including a component similar to the one we presented in this chapter. Nevertheless, I did not conduct further research on the imperfect information aspect of RTS games. I chose instead to focus on learning combat models and using abstractions to make adversarial search approaches feasible for domains with large action and state spaces.

Chapter 4

Predicting Army Combat Outcomes in StarCraft

This chapter is joint work with Sergio Poo Hernandez, Graham Erickson, Russel Greiner and Michael Buro. It was previously published [206] at the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE), 2013.

Abstract

Smart decision making at the tactical level is important for Artificial Intelligence (AI) agents to perform well in the domain of real-time strategy (RTS) games. This paper presents a Bayesian model that can be used to predict the outcomes of isolated battles, as well as predict what units are needed to defeat a given army. Model parameters are learned from simulated battles, in order to minimize the dependency on player skill. We apply our model to the game of Starcraft, with the end-goal of using the predictor as a module for making high-level combat decisions, and show that the model is capable of making accurate predictions.

4.1 Introduction

4.1.1 Purpose

Real-Time Strategy (RTS) games are a genre of video games in which players must gather resources, build structures from which different kind of troops can be trained or upgraded, recruit armies and command them in battle against opponent armies. RTS games are an interesting domain for Artificial Intelligence (AI) research because they represent well-defined complex adversarial systems and can be divided into many interesting sub-problems [28]. The current best RTS game-playing AI still performs quite poorly against human players. Therefore, the research community is focusing on developing RTS

agents to compete against other RTS agents [30]. For the purpose of experimentation, the RTS game Starcraft[1] is currently the most common platform used by the research community, as the game is considered well balanced, has a large online community of players, and has an open-source interface (BWAPI[2]).

Usually, in a RTS game, there are several different components a player needs to master in order to achieve victory. One such sub-problem is the combat scenario (usually called a battle). Each player has a known quantity of each type of unit (called an *army*) and is trying to defeat the opponent's army while keeping his own units alive. Combat is an important part of playing RTS games, as winning battles will affect the outcome of a match. We consider *1*) given two specific armies (each composed of a specified set of units of each type), predicting which will win; and *2*) given one army, specify what other army is most likely to defeat it.

4.1.2 Motivation

One successful framework for developing AI for the combat aspect of RTS games relies on alpha-beta search, where nodes are evaluated by estimating the combat outcome of two specified armies [30] (i.e., the winning player), assuming that the two players fight until one of them has no units left. One standard way to predict the outcome of such a combat is to use a simulator, where the behavior of the units is determined by deterministic scripts (e.g., attack closest unit) [44]. This is time intensive, especially as the number of units grows. The model we propose is inspired by rating systems such as the ELO ratings used in Chess [58] or TrueSkill™ [104]. By learning from battle outcomes between various combinations of units, one can predict the combat outcome of such a battle using a simple mathematical equation, rather than time consuming simulations. We are particularly interested in large scale battles, which would take a longer time to solve using such simulators.

A second challenging problem is determining an army that will have a good chance of defeating some other specified army. Given some unit combination controlled by the opponent, we want to know similar sized armies that can probably defeat it. A system that can answer this type of questions could be used by tactical planners to decide which units should be built and what unit combinations should be sent to battle the opponent. For example, if an important location (e.g., near a bridge or choke-point) is controlled by the opponent with X units (X_1 of type 1, X_2 of type 2, etc), what unit combination Y should be sent to defeat X? Even more, what properties would such a combination Y require? Maybe it needs firepower above a certain threshold, or great mobility. Currently, there are no systems that can answer such questions quickly and accurately.

[1] http://en.wikipedia.org/wiki/StarCraft

[2] http://code.google.com/p/bwapi/

4.1.3 Objectives

The objectives of this work are to develop a model that can answer these two types of questions effectively. Accuracy for the battle prediction problem (i.e., given two armies, who wins?) can be measured by the effectiveness of a trained model on match-ups set aside for testing. Accuracy for the most likely army problem (i.e., given one army, what other army is most likely to defeat it?) can be measured using play-out scripts (i.e., when fixed policies are used, does the army predicted to win, actually win?). From now on we will refer to the first question as *who wins*, and the second question as *what wins*.

The next section presents background on Bayesian Networks as applied to RTS games, and on ranking systems. In section 3 we introduce our proposed model, and explain the process of learning the model parameters. In section 4, we present the data we use to evaluate the model, along with several experiments and a discussion of the results. Future extensions and conclusions are discussed in the section 5.

4.2 Background

4.2.1 Bayesian networks

The model we present uses Bayesian techniques in the form of a Bayesian network, which is a type of Probabilistic Graphical Model (PGM) [122]. Bayesian Networks represent a set of random variables (which could be observable quantities, latent variables or unknown parameters) and their conditional dependencies using a directed acyclic graph, whose edges denote conditional dependencies. Each node has an associated probability function that, for each specific assignment to the node's direct parents, gives a probability distribution of the variable represented by that node.

Bayesian networks encode the uncertainty of the situation, allowing incompleteness (in a formal logic sense) to be transformed into uncertainty about the situation [112]. There is much uncertainty in RTS games: the opponent's state is only partially known, moves are made simultaneously, and RTS games represent complex, dynamic environments that are difficult to model completely. Even more, PGMs allow us to develop a single model that can support different types of queries. Using variables that have known values as '*evidence*', we can inquire about the (conditional) probability distribution of the remaining variable(s).

Naturally, PGMs have seen an increased popularity in the RTS domain over the past few years. Hidden Markov Models (a simple type of PGM) have been used to learn high-level strategies from data [51] and PGMs have been used to predict the opponent's opening strategy [216] and to guess the order that the opponent is building units in [218]. The same research group has also developed models that allow their RTS agent to make decisions about

where on the map it should send units to attack and with what kinds of units [220]. The model makes major simplifications about unit types and does not allow the agent to ask questions about how many of a specific unit type it should produce. [218] have also used Bayesian modeling to enable units to be individually reactive. This allows each unit to behave individually when navigating the map; moreover, during combat the unit will determine if it needs to retreat or continue fighting. The model only works for individual unit decisions, but is not used to predict an outcome between two armies, which is one of our interests.

Most recently, the same research group has clustered armies based on their unit compositions and shown how battle predictions can be made using the cluster labels [219], which effectively tackles the *who wins* problem. However, their method was shown to have a low prediction accuracy, and differs from ours in two ways. First, their model was developed on replay data (instead of simulated data) which adds the noise of different player skills to the problem. Second, their model can only be used to predict the following question: given two armies, which one wins? It cannot be used to predict an army that can defeat the opponent.

We chose to use a Bayesian network for our model, as there are many advantages to such a choice. First, we could learn the model once, then answer both types of questions simply by performing different types of queries on the same model. Secondly, defining the model structure (based on intuition and domain knowledge) helps to simplify the learning and the inference tasks. Finally, Bayesian networks allow uncertainty to be modeled explicitly (i.e., predictions are reported in form of likelihoods or probabilities).

4.2.2 Rating Systems

A problem similar to battle outcome prediction is rating/ranking – the task of attaching some numeric quantities to subjects, then arranging these subjects in a specific order, consistent with the assigned values. Rating systems have been historically used to estimate a player's skill in one-on-one matches [58], and subsequent systems even measured the uncertainty of that estimation [78].

During the last decade, rating systems have been extended for events that include more players. The most significant results were obtained by TopCoder's ranking algorithm[3] and by Microsoft's approach, TrueSkill™ [104]. The probabilistic model used by TrueSkill™ is designed to deal with players that take part in games or enter tournaments, and compete in teams of various sizes. It estimates the skills of these players after each match (or competition). The system is initialized with a prior one dimensional Gaussian distribution over each player's skills: $s \sim N(\mu, \sigma^2)$. The mean player's *true skill* is μ, where σ indicates the uncertainty of the prior. After k games, the posterior can be

[3] http://apps.topcoder.com/wiki/display/tc/Algorithm+Competition+Rating+System

computed, and we obtain μ^k, σ^k, where this σ^k will usually decrease with the number of matches, as we get more information about the respective player. This approach could prove very useful, if we treat a battle as a match between an arbitrary number of units on each side. Then, after observing a number of battles, we would have 'skills' estimates for each unit type. As each army is a 'team' of units, we can combine the 'skills' of all units in a team/army – for example, by adding them up – and use this sum to predict the outcome of any future battle.

4.3 Proposed model & Learning task

Current rating systems associate a single latent variable to each person (his skill). This may work well for predicting the outcome of a Chess match, but units in RTS games are inherently different and the outcome of a battle depends on features such as damage, attack range, hit points, armor or speed. Consequently, we need a model with multiple latent features for every unit. Besides being able to predict battle outcomes, such a model could also provide insight into why an army defeats another (e.g., army A wins because it has very high damage output while army B is lacking in the hit points attribute).

4.3.1 The Model

Using a unit's hit point and attack values, [44] propose the following evaluation function for combat games, based on the life-time damage a unit can inflict:

$$\text{LTD} = \sum_{u \in U_A} \text{Hp}(u)\text{Dmg}(u) - \sum_{u \in U_B} \text{Hp}(u)\text{Dmg}(u)$$

U_A and U_B are the units controlled by player A and B; $\text{Hp}(u)$ and $\text{Dmg}(u)$ are the hit points and damage the unit u inflicts per second. This was shown to be effective and could serve as a starting point for our model. [68] prove that in 1 vs. n units combat scenarios, there is an optimal way for the lone unit to choose its targets: to minimize its sustained damage, it should order its targets by decreasing value of $\text{Dmg}(u)/\text{Hp}(u)$.

We would like to use a similar formula to predict whether army A can win against army B. Of course, since an army has more units, we need to define composite features such as Hp(A), Hp(B), Dmg(A), Dmg(B), where for example $\text{Hp(A)} = \sum_{u \in U_A} \text{Hp}(u)$. We can then define the quantity:

$$\text{Dmg(A)/Hp(B)} - \text{Dmg(B)/Hp(A)},$$

This expression will directly influence the probability of army A winning against army B. The winning probability for army A will be higher if the offensive feature – Dmg(A) – is high, and the opponent's defense – Hp(B) – is low. Our intuition is that combining terms from both armies (such as damage

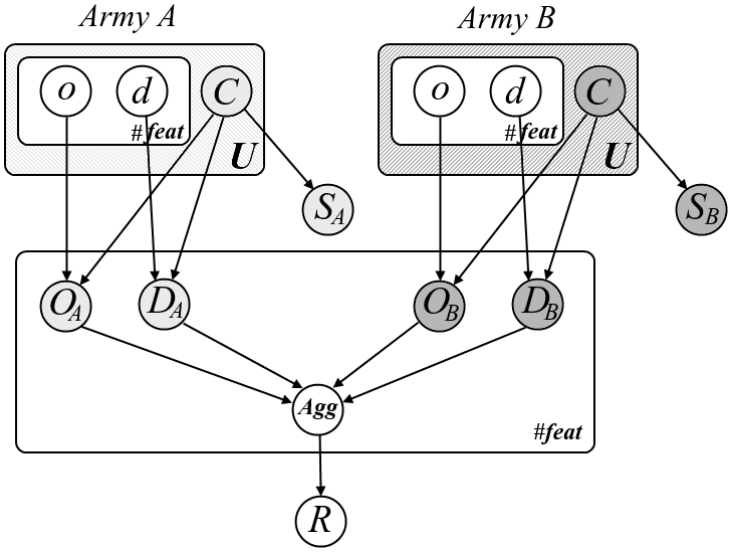


Figure 4.1: Proposed graphical model.

of first army over hit points of second army) should work well. Moreover, it allows us to generalize from damage and hit point to offensive and defensive features. That way, we could integrate into our model interactions such as damage type vs. armor, or army mobility vs. attack range by using a general term such as $(O_A/D_B - O_B/D_A)$. We assume all features to have strictly positive values. A similar idea worked very well in [199].

The resulting graphical model is shown in Figure 1. We will briefly explain each type of node. At the top we have a discrete node for each type of unit, representing the number of units (0-100) of that type in an army (node C = unit **C**ount). There are U different unit types, and so U instances of this plate for each army (for *plate notation* see [25]); there are two plates (left, right) that represent the two different armies in the modeled network. Next, for every unit there are *#feat* tuples of features, each having an offensive (node o) and defensive (node d) attribute, as described above (for a total of *2*#feat* features). For example, some offensive/defensive tuples could be {damage, hit points} or {attack range, mobility}. These nodes are one dimensional Gaussian variables.

Then, for every feature, we compute the aggregate offensive (node O) and defensive (node D) features for each army by adding all individual unit features: $O = \sum_{i=1}^{U} C_i \cdot o_i$. This is of course a simple model; here we explore whether it is sufficient. Next, for each feature, we combine the aggregate offense and defense into a single node, using formula previously introduced:

$$\mathrm{Agg}_i = (O_{A_i}/D_{B_i} - O_{B_i}/D_{A_i}).$$

There will be a total of *#feat* such nodes, which we anticipate to provide the information needed to determine the outcome of the battle. This last node (R - **R**esult) provides a value $\in [0, 1]$, corresponding to the probability that army A will defeat army B. For this, we use a sigmoid function [91] of the sum of

all combined nodes:

$$R = \frac{1}{1 + e^{-\sum_{i=1}^{F} \text{Agg}_i}}$$

There is one remaining node type (S - **S**upply), which enforces supply restrictions on armies. Supply is a value Starcraft uses to restrict the amount of units a player can command: each unit has a supply value (e.g., marine 1, zealot 2, dragoon 2, etc.), and an army's supply value is the sum of its composing units' supply values. We incorporate supply into our model to avoid trivial match-ups (e.g., 20 marines will defeat 1 marine), and to be able to ask for armies of a specific size.

We decided to start with this simple but potentially effective model structure, and focus on learning the features in an efficient manner. In future work we could investigate more complex models, and other ways of combining the features (rather than a simple sum).

4.3.2 Learning

Having specified the structure of our model, we need to learn the offensive and defensive feature values for each unit type (nodes at the top of our graphical model). Afterwards, we can start asking queries and predicting battle outcomes.

Let $M = (m_1, \ldots m_N)$ be the results of the observed matches, where each m_k is either A or B, depending on who won. We let $F = (\text{feat}_i^j)$ denote the vector of features for all unit types; feat_i^j is feature i $\in \{1, 2, \ldots 2 * \#feat\}$ of unit type j (out of U unit types). We will learn one such vector $F' = \arg\max P(M, F)$ based on observing the dataset M, as we assume that the unit features stay constant and do not change between matches (eg. a marine will always have the same damage, speed or hit points). The joint distribution of F and the N results of the matches is then

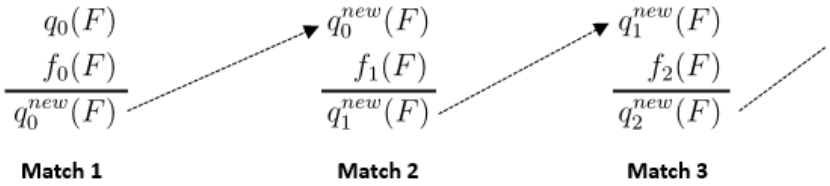
$$P(M, F) = P(F) \prod_i P(m_i|F) \text{ (m_j is the result of match j).}$$

Because exact inference is intractable (at one point we will need to compute integrals of sigmoid functions times Gaussians), we will use the core approximation technique employed by [104] in TrueSkill™ - Gaussian density filtering (GDF). This method, also known as *moment matching* or *online Bayesian learning*, is commonly used for approximating posteriors in Bayesian models [146]. Given a joint distribution over some observed variables M and hidden parameters F, it computes a Gaussian approximation q of the posterior $P(F|M)$:

$$q(F) \sim N(\mu, \sigma).$$

To use Gaussian density filtering, we need to factor the joint distribution into a product of factors $P(M, F) = \prod_i f_i$. We can choose $f_0 = p(F)$ as the prior and $f_i(F) = p(m_i|F)$ as the other factors, one for each battle. We use

the prior to initialize the posterior, and step through all the factors, updating and incorporating each one into our posterior. At every step we start with a Gaussian belief about the feature vector F, which is our current approximation $q(F)$. We update it based on the new observation's likelihood $f_i(F)$ to obtain an approximate posterior $q_i^{new}(F)$:



The exact posterior, which is difficult to compute, is

$$\hat{P}_i(F) = \frac{f_i(F)q(F)}{\int_F f_i(F)q(F)\,dF}.$$

We find the approximate posterior $q_i^{new}(F)$ by minimizing the KL divergence: $q_i^{new}(F) = \arg\min_q \mathrm{KL}(\hat{P}_i(F)||q)$, while requiring that it must be a Gaussian distribution [146]. This reduces to moment matching, hence the alternative name for this method. The $q_N^{new}(F)$ obtained after processing all factors is the final approximation we will use in our model.

4.4 Experiments

Because we are most interested in comparing armies in terms of the units that compose them, we made several simplifications of an RTS battle. Terrain or advantages caused by terrain are not considered by the model. Spell-casters and flying units are also left out. Upgraded units and upgrades at the per-unit level are not taken into account. Battles are considered to be independent events that are allowed to continue until one side is left with no remaining units. That is, we do not represent reinforcements or retreating in our model, and the outcome of one battle is unrelated to the outcome of another battle. Furthermore, only one-on-one battles (in terms of one player versus another player) are modeled explicitly.

4.4.1 Data

For the prediction problem (*who wins?*), the model's input is in the form of tuples of unit counts. Each player is represented by one tuple, which has an element for each unit type. For the current version of the model, only four different unit types are considered (here two Terran faction units - marines and firebats, and two Protoss faction units - zealots and dragoons). For each unit type, the value of the element is the number of units of that type in the

player's army. The tuples refer to the armies as they were at the start of the battle. The output of the model is a soft prediction (a probability of one of the players winning).

The input to the model for the most likely army problem (*what wins?*) is similar. Two army tuples are given, but the values in some or all of the elements of one of the armies are missing (including the supply value, which, if specified, can be used as a restriction). The output is an assignment for the missing values that corresponds to the army most likely to win the battle.

For training and testing purposes, we generated data-sets using a Starcraft battle simulator (*Sparcraft*, developed by David Churchill, UAlberta[4]). The simulator allows battles to be set up and carried out according to deterministic play-out scripts (or by decision making agents, like an adversarial search algorithm). We chose to use simulated data (as opposed to data taken from real games) because the simulator allows us to produce a large amount of data (with all kinds of different unit combinations) and to avoid the noise caused by having players of different skill and style commanding the units. This would be an interesting problem to investigate, but it is outside of the scope of this paper.

The simulations use deterministic play-out scripts. Units that are out of attack range move towards the closest unit, and units that can attack target the opponent unit with the highest damage-per-second to hit-point ratio. This policy was chosen based on its success as an evaluation policy in search algorithms [44]. Two data-sets were created: *1*) a data-set of armies of ten supply (33 different armies, 1089 different battles), and *2*) a data-set of armies of fifty supply (153 different armies, 23409 different battles).

The simulator does not have unit collision detection, which means that we can position all units of an army at the same position on the map. Using this option, we can generate two datasets for each supply limit. One dataset has the armies at opposite sides of the map in a line formation. This was an arbitrary choice and it is not affecting the model's parameters. The other dataset has all units of each army in a single fixed position at opposite sides of the map. We explored how using the single fixed position affects the accuracy of the model.

We are interested in answering two main questions: **who wins?** - given two armies, which one is more likely to win? and **what wins?** - given an opponent army, what army do we need to build in order to defeat it? Both questions will be tested on the 10 supply and 50 supply data sets.

4.4.2 Who Wins

This section describes the experiments we ran to answer the *who wins* question. First, we are interested in the model's capability to generalize, and how it performs given a number of battles for training. If there is good performance

[4]https://code.google.com/p/sparcraft/

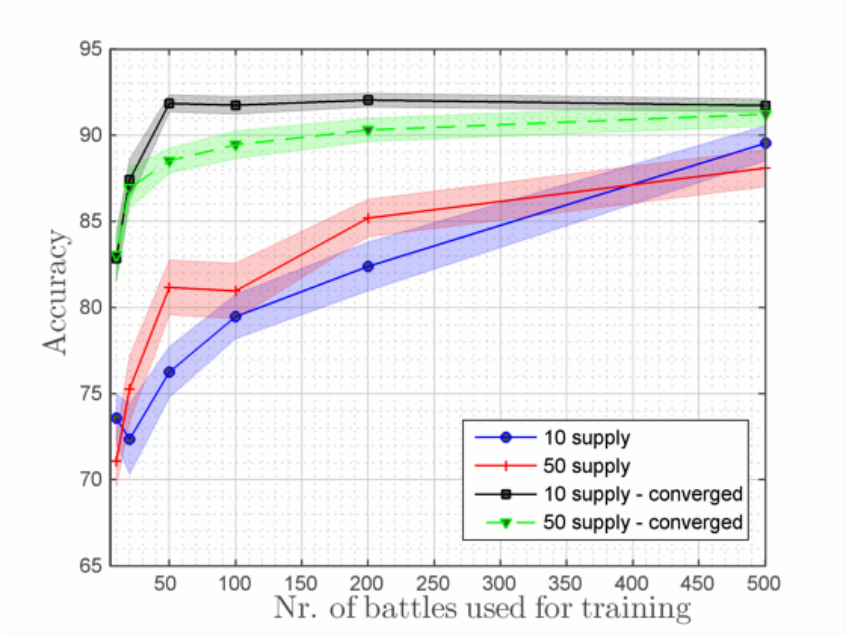


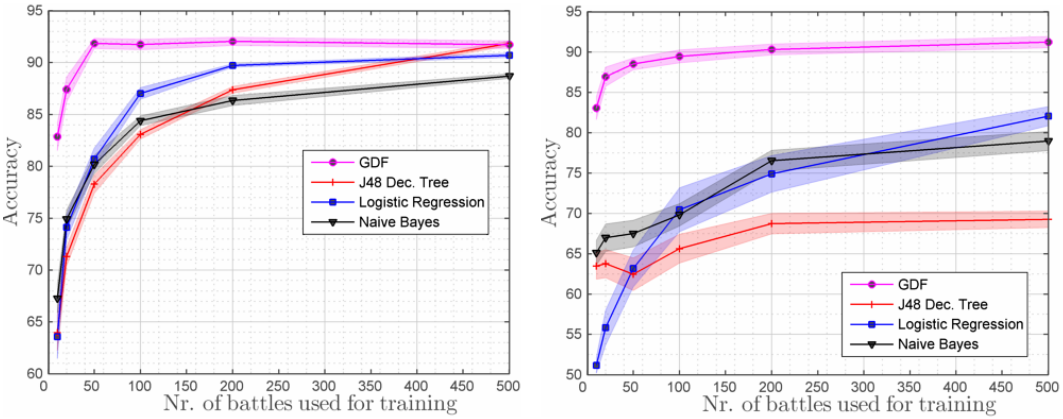
Figure 4.2: Accuracy results for *GDF - one pass* and *GDF - until convergence*, for **who wins** experiments.

even for a low number of battles, then we can train against a specific opponent, for example when playing StarCraft AI tournaments. We randomly chose 10, 20, 50, 100, 200 and 500 different battles for training, and 500 other battles to predict as a test. The accuracy is determined by how many outcomes the model is able to predict correctly. We compute the mean accuracy for 20 experiments, and show errorbars (shaded area) for one standard error on either side of the mean.

[146] notes that more passes of GDF on the same (training) data leads to significantly improved results, a tendency also confirmed by [199]. Consequently, after we process the data once, we run the same algorithm again using our approximation as a starting point. We repeat this until convergence – when the difference in successive approximations falls under a certain threshold. In our case, around 10 iterations were enough to reach the best performance. We show the results in Figure 2.

The more training data we use the better the model performs, which is what we expected. The results are very encouraging; the improvement brought by several GDF passes is obvious, and training on more than 50 battles provides almost no additional gain. In the following experiments we use this version of the algorithm, as the results are significantly better than only one GDF pass.

We compare our method with several popular algorithms, as a baseline. We use logistic regression, a naive bayes classifier, and J48 decision trees, all of which are implemented in Weka using default parameters [90]. We chose these classifiers because they are well-known and simple to implement. The results are shown in Figure 3 (for 10 supply armies) and in Figure 4 (for 50 supply armies).



(a) Accuracy results comparing GDF and three standard classifiers, for *who wins* experiments, 10 supply battles.

(b) Accuracy results comparing GDF and three standard classifiers, for *who wins* experiments, 50 supply battles.

Figure 4.3: Accuracy results for *who wins* experiments.

For 10 supply armies, all algorithms have a similar performance for training with large datasets (500 battles). However, the lower the number of battles, the better our algorithm does, in comparison. After training on 20 battles, the accuracy is better than any of the other algorithms, trained on 100 battles. When increasing the size of the armies to 50 supply, the results are even better. Even training with only 10 battles, we achieve well over 80% accuracy, better than any of the baseline algorithms do even after seeing 500 battles!

Finally, in Table 4.1 we compare different starting positions of the armies: line formation vs. all units juxtaposed at the same fixed position. In addition, a 10 supply data set was created that uses a search algorithm as the play-out policy instead of simple scripted policies. This provides better accuracy than its scripted counterpart, probably because the fights are played by '*stronger*' players. For the juxtaposed armies, the accuracy drops drops by 1-2% compared to the spread formations. We conjecture this may be because clustering

		Number of battles in training set					
Pos.	S	10	20	50	100	200	500
Line		82.83%	87.38%	91.81%	91.71%	92.00%	91.69%
Fixed		82.39%	85.96%	86.75%	89.10%	89.94%	90.00%
Fixed	✓	82.71%	87.37%	91.57%	90.84%	91.02%	90.77%
Line		83.03%	86.90%	88.48%	89.41%	90.27%	91.18%
Fixed		81.94%	86.29%	85.99%	85.12%	86.02%	85.02%

Table 4.1: Accuracy of *who wins* experiment, using different starting positions for 10 (upper section) and 50 supply armies (lower section).

range units together provides them with an increasing advantage, as the size of the armies grow. They would be able to focus fire and kill most of the oncoming opponents instantly, and they would be far more valuable in high numbers. Note, however, that our current system is not able to model this, as it computes the army features as a sum of all individual features. The model is not even aware of the unit positions, as the only inputs are the number and type of units, along with the winner.

4.4.3 What Wins

For this question we first created all possible Protoss and Terran army combinations (of two unit types) that can be built with 10 and 50 supply (44 and 204 combinations, respectively). Once trained (using the same data as the *who wins* experiment), we provide each (known) army combination as inputs. Then, the model will predict the Protoss army that is most likely to defeat (with 90% probability) the known, given army. Sometimes, if the given army is very strong, the model is not able to provide an army to defeat it. Each given answer is then tested with the simulator to verify that the predicted army wins, as expected. The accuracy of the model is measured by the percentage of instances where the predicted army actually wins.

The model has two options for predicting armies:

- With an army supply limit, the predicted army supply size must be equal to the limit.
- Without an army supply limit, the predicted army can be of any size.

The results are shown in the Table 4.2. We see that the model is able to predict a correct army that wins against an army of 50 supply 97% of the time, when no supply limit is specified. With the supply limit enabled, the accuracy drops to 87%. Against armies of 10 supply it is less accurate in both cases, and drops again, from 82% to 68%. Predicting winning 10 supply armies is the hardest task, because there are a few very strong armies, making it hard to defeat with the limited available choices of 10 supply (5 Protoss units). Therefore, the model is unable to predict an army that wins with 90% probability in almost 22% of the cases.

It is clear that with no supply limit the system is able to accurately predict an army that can win, which might happen because there are more available armies to choose from, most of which have larger supply than the opponent (and are clearly advantaged). The closer we ask the model to match the known army in supply, the worse it is at estimating a winning army, because the battles are more evenly matched.

4.5 Future Work & Conclusions

We are currently exploring several extensions:

Data	Supply limit	Predicted army		
		Wins	Loses	Unavailable
10 supply		82.5%	17.5%	0%
10 supply	✓	68.2%	9.1%	22.7%
50 supply		97.9%	0%	2.1%
50 supply	✓	87.3%	8.8%	3.9%

Table 4.2: Accuracy of *what wins* experiment, with and without supply limits for 10 and 50 supply armies.

- Currently the model is using 6 features for each unit type. We plan to increase the number of features to explore how this affects the accuracy for both questions. It would be interesting to see if adding more features increases accuracy or if it leads to overfitting.

- We are also adding more army constraints. Currently we are using the supply cost of the units in the army, but another way of limiting the army size is by the resource cost of training that army. Resources are one of the most popular metrics used to determine the army strength by other systems that work with StarCraft. Being able to enforce both supply and resource constraints would prove very useful.

- Our current model deals with only 4 unit types (marines, firebat, zealots and dragoons). We would like to expand it to use other unit types available in StarCraft. This change is critical if we want the model to be used as part of a Starcraft playing AI.

- We would like to represent additional information, such as the units' positions or number of hit points. Currently we treat all units as 'new' and having maximum life, which is not always the case.

- We work with a simulated dataset, which makes everything easier by disregarding several aspects of the game such as unit collision and the noise induced by human micromanagement skills. We would like to compare our model on *real data* (for example extracted from game replays), which is a more difficult task.

- One limitation for the current model is assuming that units have independent contributions to the battle outcome. This may hold for a few troop types, but is particularly wrong when considering units such as spell-casters, which promote interactions with other units using spells. We also miss taking into account combinations such as melee+ranged units, which are more efficient in general. We need to either consider a few smart features to take these into account, or add correlations between different types of units.

- Finally, we want to expand the questions the model can answer. We want to investigate how good the model is at estimating – given some already-built units – what units we need to add to our army in order to defeat our opponent. Expanding the model should be straightforward, as we could simply duplicate the number of unit count nodes: for each type of unit, have a node for the number of already existing units, and one for units that could potentially be built. This would also work for representing the opponent, making the difference between observed or unobserved units.

The results we obtain are very promising. Our model can very accurately predict *who wins*. The model does not do as well in estimating *what wins* but the results are positive and it might be just a matter of modifying the model by adding features that work better at estimating the answer for this question. Moreover, trying more complex ways of combining the features (rather than a simple sum) should lead us to a better understanding of the problem, and could further increase the accuracy of the model.

A StarCraft agent would greatly benefit from incorporating such a framework. Accurately predicting *who wins* could be used to avoid fighting battles against superior armies, or to determine when flow of a battle is against the player and the units should be retreated. However, sometimes losing units could potentially be strategically viable, if the armies are close in strength but the opponent's army was more expensive (either resource cost or training time). Estimating *what wins* would be used to decide what units to build in order to best counter the opponent's army. Furthermore, it could also help the build order module by making suggestions about the units needed in the future.

These are problems often encountered in the real-time strategy domain, and consequently our model would prove useful in most RTS games. We also think that it could potentially transfer to other games such as multiplayer online battle arena (MOBA) games (eg. DOTA[5]). By recording the starting hero for each of the ten players (who fight in a 5 vs. 5 match) and the outcome of the game, we could potentially find out which heroes are stronger than others (overpowered) for balancing reasons. A system that makes recommendations for which heroes to chose at the start of the game is a viable option, too.

4.6 Contributions Breakdown and Updates Since Publication

The bulk of the work in this chapter was performed by Marius Stanescu. Sergio Poo Hernandez and Graham Erickson helped with generating the data-sets for training and testing purposes, running experiments and writing the published article. Russel Greiner and Michael Buro supervised the work.

[5]https://en.wikipedia.org/wiki/Defense_of_the_Ancients

Several classical machine learning classification algorithms like Linear and Quadratic Discriminant Analysis, Support Vector Machines and kNN-based versions were used for similar purposes [182]. This approach was different in the sense that it predicted the outcome as the game progressed, and used identical armies for the two players. Location of the troops was included in the models, and experiments were run in StarCraft.

While being a promising approach, for the described model to be useful in practice there are several limitations that need to be addressed:

- the model is linear in the unit features, which does not capture well concentration of fire;
- small number of unit types tested;
- predicts the winner but not the remaining army size;
- no support for units with less than full health;
- experiments are done using simulators.

These are all fixed by research described in the next chapter. Such extensions were required for use within a higher level search algorithm, either as an evaluation function, as a forward model for combat, or both. Chapter 7 describes a method that also takes unit locations into consideration and is an even stronger evaluation function.

Chapter 5

Using Lanchester Attrition Laws for Combat Prediction in StarCraft

This chapter is joint work with Nicolas A Barriga and Michael Buro. It was previously published [202] at the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE), 2015.

Abstract

Smart decision making at the tactical level is important for Artificial Intelligence (AI) agents to perform well in the domain of real-time strategy (RTS) games. Winning battles is crucial in RTS games, and while humans can decide when and how to attack based on their experience, it is challenging for AI agents to estimate combat outcomes accurately.

A few existing models address this problem in the game of StarCraft but present many restrictions, such as not modeling injured units, supporting only a small number of unit types, or being able to predict the winner of a fight but not the remaining army. Prediction using simulations is a popular method, but generally slow and requires extensive coding to model the game engine accurately.

This paper introduces a model based on Lanchester's attrition laws which addresses the mentioned limitations while being faster than running simulations. Unit strength values are learned using maximum likelihood estimation from past recorded battles. We present experiments that use a StarCraft simulator for generating battles for both training and testing, and show that the model is capable of making accurate predictions. Furthermore, we implemented our method in a StarCraft bot that uses either this or traditional simulations to decide when to attack or to retreat. We present tournament results (against top bots from 2014 AIIDE competition) comparing the performances of the two versions, and show increased winning percentages for our

method.

5.1 Introduction

A Real-Time Strategy (RTS) game is a video game in which players gather resources and build structures from which different types of units can be trained or upgraded in order to recruit armies and command them into battle against opposing armies. RTS games are an interesting domain for Artificial Intelligence (AI) research because they represent well-defined complex adversarial environments and can be divided into many interesting sub-problems [28]. Current state of the art AI systems for RTS games are still not a match for good human players, but the research community is hopeful that by focusing on RTS agents to compete against other RTS agents we will soon reach the goal of defeating professional players [166]

For the purpose of experimentation, the RTS game StarCraft [1] is currently the most common platform used by the research community, as the game is considered well balanced, has a large online community of players, and features an open-source programming interface (BWAPI [2]).

RTS games contain different elements aspiring players need to master. Possibly the most important such component is *combat* in which each player controls an army (consisting of different types of units) and is trying to defeat the opponent's army while minimizing its own losses. Winning such battles has a big impact on the outcome of the match, and as such, combat is a crucial part of playing RTS games proficiently. However, while human players can decide when and how to attack based on their experience, it is challenging for current AI systems to estimate combat outcomes.

[41] estimate the combat outcome of two armies for node evaluation in their alpha-beta search which selects combat orders for their own troops. Similarly, [201, 200] require estimates of combat outcomes for state evaluation in their hierarchical search framework and use a simulator for this purpose. Even if deterministic scripted policies (e.g., "attack closest unit") are used for generating unit actions within the simulator [44], this process is still time intensive, especially as the number of units grows.

[206] recognize the need for a fast prediction method for combat outcomes and propose a probabilistic graphical model that, after being trained on simulated battles, can accurately predict winners. While being a promising approach, there are several limitations that still need be addressed:

- the model is linear in the unit features (i.e., the offensive score for a group of 10 marines is ten times the score for 1 marine). While this could be accurate for close-ranged (melee) fights, it severely underestimates being able to focus fire in ranged fights (this will be discussed in depth later)

[1] http://en.wikipedia.org/wiki/StarCraft
[2] http://code.google.com/p/bwapi/

- their model deals with only 4 unit types so far, and scaling it up to all StarCraft units might induce training problems such as overfitting and/or accuracy reduction
- it only predicts the winner but not the remaining army size
- all units are treated as having maximum hit points, which does not happen often in practice; there is no support for partial hit points
- all experiments are done on simulated data, and
- correlations between different unit types are not modeled.

In this paper we introduce a model that addresses the first five limitations listed above, and we propose an extension (future work) to tackle the last one. Such extensions are needed for the model to be useful in practice (e.g., for speeding up hierarchical search and adjusting to different opponents by learning unit strength values from past battles).

We proceed by first discussing current combat game state evaluation techniques and Lanchester's battle attrition laws. We then show how they can be extended to RTS games and how the new models perform experimentally in actual StarCraft game play. We finish with ideas on future work in this area.

5.2 Background

As mentioned in the previous section, the need for a fast prediction method for combat outcomes has already been recognized [206]. The authors propose a probabilistic graphical model that, after being trained on simulated battles, can accurately predict the winner in new battles. Using graphical models also enables their framework to output unit combinations that will have a good chance of defeating some other specified army (i.e., given one army, what other army of a specific size is most likely to defeat it?).

We will only address the first problem here: single battle prediction. We plan to use our model for state evaluation in a hierarchical search framework similar to those described in [200] and [232]. Consequently, we focus on speed and accuracy of predicting the remaining army instead of only the winner. Generating army combinations for particular purposes is not a priority, as the search framework will itself produce different army combinations which we only need to evaluate against the opposition.

Similarly to [206], our model will learn unit feature weights from past battles, and will be able to adjust to different opponents accordingly. We choose maximum likelihood estimation over a Bayesian framework for speed and simplicity. Incorporating Lanchester equations in a graphical model would be challenging, and any complex equation change would require the model to be redesigned. The disadvantages are that potentially more battles will be

needed to reach similar prediction accuracies and batch training must be used instead of incrementally updating the model after every battle.

There are several existing limitations our model will address:

- better representation of ranged weapons by unit group values depending exponentially on the number of units instead of linearly,
- including all StarCraft unit types,
- adding support for partial hit points for all units involved in the battle, and
- predicting the remaining army of the winner.

5.2.1 Lanchester Models

The seminal contribution of Lanchester to operations research is contained in his book "Aircraft in Warfare: The Dawn of the Fourth Arm" [132]. He starts by justifying the need for such models with an example: consider two identical forces of 1000 men each; the Red force is divided into two units of 500 men each which serially engage the single (1000 man) Blue force. According to the Quadratic Lanchester model (introduced below), the Blue force completely destroys the Red force with only moderate loss (e.g., 30%) to itself, supporting the "concentration of power" axiom of war that states that forces are not to be divided. The possibility of equal or nearly equal armies fighting and resulting in relatively large surviving forces for the winner army are one of the interesting aspects of war simulation based games.

Lanchester equations represent simplified combat models: each side has identical soldiers, and each side has a fixed strength (no reinforcements) which governs the proportion of enemy soldiers killed. Range, terrain, movement, and all other factors that might influence the fight are either abstracted within the parameters or ignored entirely. Fights continues until the complete destruction of one force (which Lanchester calls a "conclusion"). The equations are valid until one of the army sizes is reduced to 0.

Lanchester's Linear Law is given by the following differential equations:

$$\frac{dA}{dt} = -\beta AB \quad \text{and} \quad \frac{dB}{dt} = -\alpha BA \ ,$$

where t denotes time and A, B are the force strengths (number of units) of the two armies assumed to be functions of time. Parameters α and β are attrition rate coefficients representing how fast a soldier in one army can kill a soldier in the other. The equation is easier to understand if one thinks of β as the relative strength of soldiers in army B; it influences how fast army A is reduced. The pair of differential equations above may be combined into one equation by removing time as a variable:

$$\alpha(A - A_0) = \beta(B - B_0) \ ,$$

where A_0 and B_0 represent the initial forces. This is called a *state solution* to Lanchester's differential equation system that does not explicitly depend on time). The origin of the term *linear law* is now apparent because the last equation describes a straight line.

Lanchester's Linear Law applies when one soldier can only fight one other soldier at a time. If one side has more soldiers some of them will not always be fighting as they wait for an opportunity to attack. In this setting, the casualties suffered by both sides are proportional to the number actually fighting and the attrition rates. If $\alpha = \beta$, then the above example of splitting a force into two and fighting the enemy sequentially will have the same outcome as without splitting: a draw. This was originally called *Lanchester's Law of Ancient Warfare*, because it is a good model for battles fought with edge weapons.

Lanchester's Square Law is given by:

$$\frac{dA}{dt} = -\beta B \quad \text{and} \quad \frac{dB}{dt} = -\alpha A \ .$$

In this case, the state solution is

$$\alpha(A^2 - {A_0}^2) = \beta(B^2 - {B_0}^2) \ .$$

Increases in force strength are more important than for the linear law, as we can see from the concentration of power example. The squared law is also known as *Lanchester's Law of Modern Warfare* and is intended to apply to ranged combat, as it quantifies the value of the relative advantage of having a larger army. However, the squared law has nothing to do with range – what is really important is the rate of acquiring new targets. Having ranged weapons generally lets your soldiers engage targets as fast as they can shoot, but with a sword or a pike to which the Linear Law applies one would have to first locate a target and then move to engage it.

The general form of the attrition differential equations is:

$$\frac{dA}{dt} = -\beta A^{2-n} B \quad \text{and} \quad \frac{dB}{dt} = -\alpha B^{2-n} A \ ,$$

where n is called the *attrition order*. We have seen previously that for $n = 1$, the resulting attrition differential equations give rise to what we know as Lanchester's Linear Law, and to the Lanchester's Square Law for $n = 2$. As expected, the state solution is

$$\alpha(A^n - {A_0}^n) = \beta(B^n - {B_0}^n) \ .$$

The exponent which is called attrition order represents the advantage of a higher rate of target acquisition and applies to the size of the forces involved in combat, but not to the fighting effectiveness of the forces which is modeled by attrition coefficients α and β. The higher the attrition order, the faster any advantage an army might have in combat effectiveness is overcome by numeric

superiority. This is the equation we use in our model, as our experiments suggest that for StarCraft battles an attrition order of ≈ 1.56 works best on average, if we had to choose a fixed order for all possible encounters.

The Lanchester Laws we just discussed have several limitations we need to overcome to apply them to RTS combat, and some extensions (presented in more detail in the following section) are required:

- we must account for the fact that armies are comprised of different RTS game unit types and
- currently soldiers are considered either dead or alive, while we need to take into account that RTS game units can enter the battle with any fraction of their maximum hit points.

5.3 Lanchester Model Extensions for RTS Games

The state solution for the Lanchester general law can be rewritten as

$$\alpha A^n - \beta B^n = \alpha {A_0}^n - \beta {B_0}^n = k \ .$$

The constant k depends only on the initial army sizes A_0 and B_0. Hence, for prediction purposes, if $\alpha {A_0}^n > \beta {B_0}^n$ then P_A wins the battle. If we note A_f and B_f to be the final army sizes, then $B_f = 0$ and $\alpha {A_0}^n - \beta {B_0}^n = \alpha {A_f}^n - 0$ and we can predict the remaining victorious army size A_f.

To use the Lanchester laws in RTS games, a few extensions have to be implemented. Firstly, it is rarely the case that both armies are composed of a single unit type. We therefore need to be able to model heterogeneous army compositions. To this extent, we replace army effectiveness α with an average value α_{avg}. Assuming that army A is composed of N types of units, then

$$\alpha_{avg} = \frac{\sum_{i=1}^{N} a_i \alpha_i}{A} = \frac{\sum_{j=1}^{A} \alpha_j}{A} \ ,$$

where A is the total number of units, a_i is the number of units of type i and α_i is their combat effectiveness. Alternatively, we can sum over all individual units directly, α_j corresponding to unit j.

Consequently, predicting battle outcomes will require a combat effectiveness (we can also call it unit strength for simplicity) for each unit type involved. We start with a default value

$$\alpha_i = \mathrm{dmg}(i)\mathrm{HP}(i) \ ,$$

where $\mathrm{dmg}(i)$ is the unit's damage per frame value and $\mathrm{HP}(i)$ its maximum number of hit points. Later, we aim to learn $\boldsymbol{\alpha} = \{\alpha_1, \alpha_2, \ldots\}$ by training on recorded battles.

The other necessary extension is including support for injured units. Let us consider the following example: army A consists of one marine with full health, while army B consists of two marines with half the hit points remaining. Both the model introduced by [206] and the life-time damage (LTD) evaluation function proposed by [126]

$$\text{LTD2} = \sum_{u \in U_A} \text{HP}(u)\text{dmg}(u) - \sum_{u \in U_B} \text{HP}(u)\text{dmg}(u)$$

would mistakenly predict the result as a draw. The authors also designed the life-time damage-2 (LTD2) function which departs from linearity by replacing $\text{HP}(u)$ with $\sqrt{\text{HP}(u)}$ and will work better in this case.

In the time a marine deals damage equal to half its health, army B will kill one of army A's marines, but would also lose his own unit, leaving army A with one of the two initial marines intact, still at half health. The advantage of focusing fire becomes even more apparent if we generalize to n marines starting with $1/n$ health versus one healthy marine. Army A will only lose one of its n marines, assuming all marines can shoot at army B's single marine at the start of the combat. This lopsided result is in stark contrast to the predicted draw.

Let's model this case using Lanchester type equations. Denoting the attrition order with o, the combat effectiveness of a full health marine with m and that of a marine with $1/n$ health as m_n, we have:

$$n^o m_n - 1^o m = (n-1)^o m_n \implies m_n = \frac{m}{n^o - (n-1)^o}$$

If we choose an attrition order between the linear ($o = 1$) and the square ($o = 2$) laws, $o = 1.65$ for example, then $m_2 = m/2.1383$, $m_3 = m/2.9887$ and $m_4 = m/3.7221$. Intuitively picking the strength of an injured marine to be proportional with its current health $m_n = m/n$ is close to these values, and would lead to extending the previous strength formula for an individual unit like so:

$$\alpha_i = \text{dmg}(i)\text{HP}(i) \cdot \frac{\text{currentHP}(i)}{\text{HP}(i)} = \text{dmg}(i)\text{currentHP}(i).$$

5.3.1 Learning Combat Effectiveness

For predicting the outcome of combat C between armies A and B we first compute the estimated army remainder score μ_C using the generalized Lanchester equation:

$$\mu_C = \alpha_C A^o - \beta_C B^o$$

From army A's perspectives μ is a positive value if army A wins, 0 in case of a draw, and negative otherwise. As previously mentioned, experiments using simulated data suggest that $o = 1.56$ yields the best accuracy, if we had

to choose a fixed order for all possible encounters. Fighting the same combat multiple times might lead to different results depending on how players control their units, and we choose a Gaussian distribution to model the uncertainty of the army remainder score r:

$$P_C(r) = N(r; \mu_C, \sigma^2) \ ,$$

where σ is a constant chosen by running experiments. Just deciding which player survives in a small scale fight where units cannot even move is PSPACE-hard in general [68]. Hence, real-time solutions require approximations and/or abstractions. Choosing a Gaussian distribution for modeling army remainder score is a reasonable candidate which will keep computations light.

Let us now assume that we possess data in the form of remaining armies A_f and B_f (either or both can be zero) from a number of combats $\boldsymbol{C} = \{C_1, C_2, \ldots, C_n\}$. A data-point C_i consists of starting army values A_i, B_i and final values A_{if}, B_{if}. We compute the remainder army score R_i using the Lanchester equation:

$$R_i = \alpha_{C_i} A_{if}^o - \beta_{C_i} B_{if}^o$$

This enables us to use combat results for training even if no side is dead by the end of the fight.

Our goal is to estimate the effectiveness values α_i and β_i for all encountered unit types and players. The distinction needs to be made, even if abilities of a marine are the same for both players. If the player in charge of army A is more proficient at controlling marines then α_{marine} should be higher than β_{marine}.

The likelihood of $\{\boldsymbol{\alpha}, \boldsymbol{\beta}\}$ given $\boldsymbol{C}$ and $\boldsymbol{R} = \{R_1, R_2, \ldots, R_n\}$ is used for approximating the combat effectiveness; the maximum likelihood value can then be chosen as an estimate. The computation time is usually quite low using conjugate gradients, for example, and can potentially be done once after several games or even at the end of a game.

If we assume that the outcomes of all battles are independent of each other and the probability of the data given the combat effectiveness values is

$$P(\boldsymbol{R}|\boldsymbol{C}, \{\boldsymbol{\alpha}, \boldsymbol{\beta}\}) = \prod_i N(R_i; \mu_{C_i}, \sigma^2) \ ,$$

then we can express the log likelihood

$$\mathcal{L}(\{\boldsymbol{\alpha}, \boldsymbol{\beta}\}) = \sum_i \log N(R_i; \mu_{C_i}, \sigma^2) \ .$$

The maximum likelihood value can be approximated by starting with some default parameters, and optimizing iteratively until we are satisfied with the results. We use a gradient ascent method, and update with the derivatives of the log likelihood with respect to the combat effectiveness values. Using a Gaussian distributions helps us to keep the computations manageable.

To avoid overfitting we modify the error function we are minimizing by using a regularization term:

$$Err = -\mathcal{L}(\{\boldsymbol{\alpha}, \boldsymbol{\beta}\}) + \gamma Reg(\{\boldsymbol{\alpha}, \boldsymbol{\beta}\})$$

If we want to avoid large effectiveness values for example, we can pick $Reg = \sum_i \alpha_i^2 + \sum_i \beta_i^2$. We chose

$$Reg = \sum_i (\alpha_i - d_i)^2 + \sum_i (\beta_i - d_i)^2 \ ,$$

where d_i are the default values computed in the previous subsection using basic unit statistics. In the experiments section we show that these estimates already provide good results. The γ parameter controls how close the trained effectiveness values will be to these default values.

5.4 Experiments and Results

To test the effectiveness of our models in an actual RTS game (StarCraft) we had to simplify actual RTS game battles. Lanchester models do not take into account terrain features that can influence battle outcomes. In addition, upgrades of individual units or unit types are not yet considered, but could later be included using *new*, virtual units (e.g., a dragoon with range upgrade is a different unit than a regular dragoon). However, that would not work for regular weapon/armor upgrades, as the number of units would increase beyond control. For example, the Protoss faction has 3 levels of weapon upgrades, 3 of shields and 3 of armor, so considering all combinations would add 27 versions for the same unit type.

Battles are considered to be independent events that are allowed to continue no longer than 800 frames (around 30 seconds game time), or until one side is left with no remaining units, or until reinforcements join either side. Usually StarCraft battles do not take longer than that, except if there is a constant stream of reinforcements or one of the players keeps retreating, which is difficult to model.

5.4.1 Experiments Using Simulator Generated Data

We start by testing the model with simulator generated data, similarly to [206]. The authors use four different unit types (marines and firebats from the Terran faction, zealots and dragoons from the Protoss faction), and individual army sizes of up to population size 50 (e.g., marines count 1 towards the population count, and zealots and dragoons count 2, etc.). For our experiments, we add three more unit types (vultures, tanks and goliaths) and we increase the population size from 50 to 100.

The model input consists of two such armies, where all units start with full health. The output is the predicted remaining army score of the winner, as the loser is assumed to fight to death. We compare against the *true* remaining army score, obtained after running the simulation.

For training and testing at this stage, we generated battles using a StarCraft battle simulator (*SparCraft*, developed by David Churchill, UAlberta[3]). The simulator allows battles to be set up and carried out according to deterministic play-out scripts, or by search-based agents. We chose the simple yet effective *attack closest* script, which moves units that are out of attack range towards the closest unit, and attacks units with the highest damage-per-second to hit point ratio. This policy, which was also used in [206], was chosen based on its success as an evaluation policy in search algorithms [44]. Using deterministic play-out scripts eliminates noise caused by having players of different skill or playing style commanding units.

We randomly chose 10, 20, 50, 100, 200, and 500 different battles for training, and a test set of 500 other battles to predict outcomes. The accuracy is determined by how many outcomes the model is able to predict correctly. We show the results in Table 5.1, where we also include corresponding results of the Bayesian model from [206] for comparison. The datasets are not exactly the same, and by increasing the population limit to 100 and the number of unit types we increase the problem difficulty.

The results are very encouraging: our model outperforms the Bayesian predictor on a more challenging dataset. As expected, the more training data, the better the model performs. Switching from 10 battles to 500 battles for training only increases the accuracy by 3.3%, while in the Bayesian model it accounts for a 8.2% increase. Moreover, for training size 10 the Lanchester model is 6.8% more accurate, but just 2% better for training size 500. Our model performs better than the Bayesian model on small training sizes because we start with already good approximations for the unit battle strengths, and the regularization prevents large deviations from these values.

[3]https://code.google.com/p/sparcraft/

Table 5.1: models, for different training sets sizes. Testing was done by predicting outcomes of 500 battles in all cases. Values are winning percent averages over 20 experiments.

	Number of battles in training set					
Model	10	20	50	100	200	500
Lanchester	89.8	91.0	91.7	92.2	93.0	93.2
Bayesian	83.0	86.9	88.5	89.4	90.3	91.2

5.4.2 Experiments in Tournament Environment

Testing on simulated data validated our model, but ultimately we need to assess its performance in the actual environment it was designed for. For this purpose, we integrated the model into UAlbertaBot, one of the top bots in recent AIIDE StarCraft AI competitions [4]. UAlbertaBot is an open source project for which detailed documentation is available online[5]. The bot uses simulations to decide if it should attack the opponent with the currently available units — if a win is predicted — or retreat otherwise. We replaced the simulation call in this decision procedure by our model's prediction.

UAlbertaBot's strategy is very simple: it only builds zealots, a basic Protoss melee unit, and tries to rush the opponent and then keeps the pressure up. This is why we do not expect large improvements from using Lanchester models, as they only help to decide to attack or to retreat. More often than not this translates into waiting for an extra zealot or attacking with one zealot less. This might make all the difference in some games, but using our model to decide what units to build, for example, could lead to bigger improvements. In future work we plan to integrate this method into a search framework and a build order planner such as [124].

When there is no information on the opponent, the model uses the default unit strength values for prediction. Six top bots from the last AIIDE competition[6] take part in our experiments: IceBot (1^{st}), LetaBot (3^{rd}), Aiur (4^{th}), Xelnaga (6^{th}), original UAlbertaBot (7^{th}), and MooseBot (9^{th}). Ximp (2^{nd}) was left out because we do not win any games against it in either case. It defends its base with photon cannons (static defense), then follows up with powerful air units which we cannot defeat with only zealots. Skynet (5^{th}) was also left out because against it UAlbertaBot uses a hard-coded strategy which bypasses the attack/retreat decision and results in a 90% win rate.

Three tournaments were run: 1) our bot using simulations for combat prediction, 2) using the Lanchester model with default strength values, and 3) using a new set of values for each opponent obtained by training on 500 battles for that particular match-up. In each tournament, our bot plays 200 matches against every other bot.

To train model parameters, battles were extracted from the previous matches played using the default weights. A battle is considered to start when any of our units attacks or is attacked by an enemy unit. Both friendly and opposing units close to the attacked unit are logged with their current health as the starting state of the battle. Their health (0 if dead) is recorded again at the end of the battle – when any of the following events occurs:

- one side is left with no remaining units,
- new units reinforce either side, or

[4] http://webdocs.cs.ualberta.ca/~cdavid/starcraftaicomp/

[5] https://github.com/davechurchill/ualbertabot/wiki

[6] http://webdocs.cs.ualberta.ca/~scbw/2014/

Table 5.2: Tournament results against 6 top bots from AIIDE 2014 competition. Three tournaments are played, with different options for the attack/retreat decision. In the first our bot uses simulations, and in the second the Lanchester model with default strength values. In the third we use battles from the second tournament for training and estimating new strength values. Winning percentages are computed from 200 games for each match-up, 20 each on 10 different maps.

	UAB	Xelnaga	Aiur	MooseBot	IceBot	LetaBot	**Avg.**
Sim.	60.0	79.0	84.0	65.5	19.5	57.0	**60.8**
Def.	64.5	81.0	80.5	69.0	22.0	66.5	**63.9**
Train	69.5	78.0	86.0	93.0	23.5	68.0	**69.7**

- 800 frames have passed since the start of the battle.

There are some instances in which only a few shots are fired and then one of the players keeps retreating. We do not consider such cases as proper battles. For training we require battles in which both players keep fighting for a sustained period of time. Thus, we removed all fights in which the total damage was less than 80 hit points (a zealot has 160 for reference) and both sides lost less than 20% of their starting total army hit points. We obtained anywhere from 5 to 30 battles per game, and only need 500 for training.

Results are shown in Table 5.2. Our UAlbertaBot version wins 60% of the matches against the original UAlbertaBot because we have updated the new version with various fixes that mainly reduce the number of timeouts and crashes, especially in the late game.

On average, the Lanchester model with learned weights wins 6% more games than the same model with default strength values, which is still 3% better than using simulations. It is interesting to note that the least (or no) improvement occurs in our best match-ups, where we already win close to 80% of the games. Most of such games are lopsided, and one or two extra zealots do not make any difference to the outcome. However, there are bigger improvements for the more difficult match-ups, which is an encouraging result.

The only exception is IceBot, which is our worst enemy among the six bots we tested against. IceBot uses bunkers to defend which by themselves do not attack but can load up to four infantry units which receive protection and a range bonus. We do not know how many and what infantry units are inside, and the only way to estimate this is by comparing how much damage our own units take when attacking it. These estimates are not always accurate, and furthermore, IceBot also sends workers to repair the bunkers. Consequently, it is very difficult to estimate strength values for bunkers, because it depends on what and how many units are inside, and if there are workers close by which can (and will) repair them. Because UAlbertaBot only builds zealots and constantly attacks, if IceBot keeps the bunkers alive and meanwhile builds

other, more advanced units, winning becomes impossible. The only way is to destroy the bunkers early enough in the game. We chose to adapt our model by having five different combat values, one for empty bunker (close to zero), and four others for bunker with one, two, three or four marines inside. We still depend on good damage estimations for the loaded units and we do not take into account that bunkers become stronger when repaired, which is a problem we would like to address in future work.

5.5 Conclusions and Future Work

In this paper we have introduced and tested generalizations of the original Lanchester models to adapt them to making predictions about the outcome of RTS game combat situations. We also showed how model parameters can be learned from past combat encounters, allowing us to effectively model opponents' combat strengths and weaknesses. Pitted against some of the best entries from a recent StarCraft AI competition, UAlbertaBot with its simulation based attack-retreat code replaced by our Lanchester equation based prediction, showed encouraging performance gains.

Because even abstract simulations in RTS games can be very time consuming, we speculate that finding accurate and fast predictors for outcomes of sub-games – such as choosing build orders, combat, and establishing expansions – will play an important role in creating human-strength RTS game bots when combined with look-ahead search. Following this idea, we are currently exploring several model extensions which we briefly discuss in the following paragraphs.

A common technique used by good human players is to snipe off crucial or expensive enemy units and then retreat, to generate profit. This is related to the problem of choosing which unit type to target first from a diverse enemy army, a challenge not addressed much in current research. Extending the current Lanchester model from compounding the strength of all units into an average strength to using a matrix which contains strength values for each own unit versus each opponent unit might be a good starting point. This extension would enable bots to kill one type of unit and then retreat, or to deal with a unit type which is a danger to some of its other units. For instance, in some cases ranged units are the only counter unit type to air units and should try to destroy all fliers before attacking ground units.

A limitation of the current model is assuming that units have independent contributions to the battle outcome. This may hold for a few unit types, but is particularly wrong when considering units that promote interactions with other units, such as medics which can heal other infantry, or workers that can repair bunkers. We could take some of these correlations into account by considering groups of units as a new, virtual unit and trying to estimate its strength.

Another limitation of our prediction model is that it completely ignores unit positions, and only takes into account intrinsic unit properties. An avenue for further research is to expand the model to take into account spatial information, possibly by including it into the combat effectiveness values.

Lastly, by comparing the expected outcome and the real result of a battle, we could possibly identify mistakes either we or the opponent made. AI matches tend to be repetitive, featuring many similar battles. Learning to adjust target priorities or to change the combat scripts to avoid losing an early battle would make a big difference.

5.6 Contributions Breakdown and Updates Since Publication

Most of the work in this chapter was performed by Marius Stanescu. Nicolas A. Barriga helped with setting up expriments, and Michael Buro supervised the work. A tutorial version of the work described in this chapter has also been published as a chapter in Game AI Pro 3, a book targeted at industry professionals [203].

Since publication, there were two research endeavors that tackled similar problems and used Lanchester-inspired models. Firstly, there is an algorithm that uses Lanchester-like equations to generate fire support plans in a simulated combat environment [92]. The authors focus on concepts that are present in modern military doctrine but are missing in most RTS games, such as tactical risk management and suppressive firepower effects. Their model leverages these concepts and is slightly different than our approach. They consider the defenders all but invulnerable until assaulted and thus use one-sided versions of the Lanchester equations, while we use a two-sided approach.

Secondly, a more extensive version of our work was presented in a journal paper [234]. Besides predicting the winner or the final state of a combat, the authors run experiments for predicting combat durations and simulating partial combats (i.e., not fight to the end). They focus on predicting which units survive for heterogeneous army compositions, and on the forward model aspect and utility of the algorithms. They propose three such models:

- *Target-Selection Lanchester's Square Law* which is similar to ours, with the addition of a target selection policy used at the end of the simulation to choose the surviving units;
- *Sustained DPF* which simplifies combat by assuming that the inflicted army damage does not decrease over time during the combat, but it models which units can attack each other in more detail than the first algorithm (e.g., ground vs. air units);
- *Decreasing DPF* which takes into account attrition and model the reduced damage an army can inflict as units die.

For predicting the winner the first model shows the best results, similar to ours. However direct comparisons are difficult as the datasets are different.

The combat model presented in this chapter was used as an evaluation function in subsequent research presented in Chapter 8, in which it led to stronger results compared to other popular heuristics used at the time. To improve the performance of high level search algorithms even further, we proposed evaluation functions that look at the entire game state instead of just combat, and take spatial details into account. Designing and choosing features for such approaches is difficult and ultimately we decided to learn these functions automatically using convolutions and neural networks, which initiated research described in Chapter 7.

Chapter 6

Hierarchical Adversarial Search Applied to Real-Time Strategy Games

This chapter is joint work with Nicolas A. Barriga and Michael Buro. It was previously published [200] at the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE), 2014.

Abstract

Real-Time Strategy (RTS) video games have proven to be a very challenging application area for artificial intelligence research. Existing AI solutions are limited by vast state and action spaces and real-time constraints. Most implementations efficiently tackle various tactical or strategic sub-problems, but there is no single algorithm fast enough to be successfully applied to big problem sets (such as a complete instance of the StarCraft RTS game). This paper presents a hierarchical adversarial search framework which more closely models the human way of thinking – much like the chain of command employed by the military. Each level implements a different abstraction – from deciding how to win the game at the top of the hierarchy to individual unit orders at the bottom. We apply a 3-layer version of our model to SparCraft – a StarCraft combat simulator – and show that it outperforms state-of-the-art algorithms such as Alpha-Beta, UCT, and Portfolio Search in large combat scenarios featuring multiple bases and up to 72 mobile units per player under real-time constraints of 40 ms per search episode.

6.1 Introduction

Real-Time Strategy (RTS) games are a genre of video games in which players gather resources, build structures from which different types of troops can be

trained or upgraded, recruit armies, and command them in battle against opponent armies. RTS games are an interesting domain for Artificial Intelligence (AI) research because they represent well-defined complex adversarial decision problems and can be divided into many interesting and computationally hard sub-problems [28].

The best AI systems for RTS games still perform poorly against good human players [30]. Hence, the research community is focusing on developing RTS agents to compete against other RTS agents to improve the state-of-the-art. For the purpose of experimentation, the RTS game StarCraft: Brood War [21] has become popular because it is considered well balanced, has a large online community of players, and has an open-source interface – BWAPI, [102] – which allows researchers to write programs to play the full game. Several StarCraft AI competitions are organized every year [30]. Such contests have sparked increased interest in RTS game AI research and many promising agent frameworks and algorithms have already emerged. However, no unified search approach has yet been developed for a full RTS game such as StarCraft, although the research community is starting to tackle the problem of global search in smaller scale RTS games [37, 180, 161]. Existing StarCraft agents rely on a combination of search and machine learning for specific sub-problems (build order [40], combat [43], strategy selection [215]) and hard-coded expert behaviour.

6.1.1 Motivation and Objectives

Even though the structure of most RTS AI systems is complex and comprised of many modules for unit control and strategy selection [249, 42, 220], none comes close to human abstraction, planning, and reasoning abilities. These independent modules implement different AI mechanisms which often interact with each other in a limited fashion.

We propose a unified perspective by defining a *multi-level abstraction framework* which more closely models the human way of thinking – much like the chain of command employed by the military. In real life a top military commander does not concern himself with the movements of individual soldiers, and it is not efficient for an RTS AI to do that, either. A hierarchical structure can save considerable computing effort by virtue of hierarchical task decomposition. The proposed AI system partitions its forces and resources to a number of entities (we may call commanders) – each with its own mission to accomplish. Moreover, each commander could further delegate tasks in a similar fashion to sub-commanders, groups of units or even individual units. More similar to the human abstraction mechanism, this flexible approach has a great potential of improving AI strength.

One way of implementing such a layered framework is to have each layer playing the game at its own abstraction level. This means we would have to come up with both the abstractions and a new game representation for

each level. Both are difficult to design and it is hard to prove that they actually model the real game properly. Another way, which we introduce in this paper, is to partition the game state and to assign objectives to individual partition elements, such that at the lowest level we try to accomplish specific goals by searching smaller state spaces, and at higher levels we search over possible partitions and objective assignments. Our approach comes with an additional benefit: the main issue that arises when approximating optimal actions in complex multi-agent decision domains is the combinatorial explosion of state and action spaces, which renders classic exhaustive search infeasible. For instance, in StarCraft, players can control up to 200 mobile units which are located on maps comprised of up to 256×256 tiles, possibly leading to more than $10^{1,926}$ states and 10^{120} available actions [166].

The hierarchical framework we propose has the advantage of reducing the search complexity by considering only a meaningful subset of the possible interactions between game objects. While optimal game play may not be achieved by this kind of abstraction, search-based game playing algorithms can definitely be more efficient if they take into account that *in a limited time frame each agent interacts with only a small number of other agents* [138].

6.2 Background and Related Work

In implementing a hierarchical framework we extend state-of-the-art RTS unit combat AI systems that use Alpha-Beta, UCT, or Portfolio Search [42, 44, 43] and focus only on one abstraction level, namely planning combat actions at the unit level. By contrast, our proposed search algorithm considers multiple levels of abstractions. For example, first level entities try to partition the forces into several second level entities – each with its own objective.

Similar approaches have been proposed by [151] and [177]. The latter implements a framework to bridge the gap between strategy and individual unit control. It conceptualizes RTS group level micromanagement as a multi-agent task allocation problem which can in principle be integrated into any layered RTS AI structure. However, while these papers only deal with the bottom level, we target several abstraction levels at the same time. Our goal is to interleave the optimization of direct unit control with strategic reasoning. Next, we look at competing algorithms designed for other problem domains whose large complexity render complete search impractical.

6.2.1 Multi-Agent Planning

A similar problem has been considered in [138]. The authors substitute global search considering all agents with multiple searches over agent subsets, and then combine the results to form a global solution. The resulting method is called Agent Subset Adversarial Search (ASAS). It was proved to run in polynomial time in the number of agents as long as the size of the subsets is

limited. ASAS does not require any domain knowledge, comparing favourably to other procedural approaches such as *hierarchical task networks* (HTNs), and greatly improves the search efficiency at the expense of a small decrease of solution quality. It works best in domains in which a relatively small number of agents can interact at any given time. Because we aim our framework specifically towards RTS games, avoiding using domain knowledge would be wasteful. Hence, we introduce a procedural component in the algorithm, in the form of objectives that each subset tries to accomplish. This will allow our algorithm to make abstract decisions at a strategic level, which should further improve the search efficiency compared to the ASAS method.

6.2.2 Goal Driven Techniques

Heuristic search algorithms such as Alpha-Beta search and A* choose actions by looking ahead, heuristically evaluating states, and propagating results. By contrast, HTNs implement a goal-driven method that decomposes goals into a series of sub-goals and tries to achieve these. For choosing a move, goals are expanded into sub-goals at a lower level of abstraction and eventually into concrete actions in the environment. In [197], the authors successfully deploy HTNs to play Contract Bridge. Using only a small number of operators proves sufficient for describing relevant plays (finessing, ruffing, cross-ruffing, etc.). HTNs have also been successfully applied to Go [247]. The advantage is that Go knowledge (e.g., in books) is usually expressed in a form appropriate for encoding goal decompositions by using a rich vocabulary for expressing reasons for making moves. However, HTNs generally require significant effort for encoding strategies as goal decompositions, and in some domains such as RTS games this task can be very difficult. For example, [144] uses HTNs to build a hierarchical architecture for Infantry Serious Gaming. Only a small set of very simple actions are supported, such as monitoring and patrolling tasks, and we can already see that the planning domain would become quite convoluted with the addition of more complex behaviour.

6.2.3 Goal Based Game Tree Search

Goal based game tree search (GB-GTS) [139] is an algorithm specifically designed for tackling the scalability issues of game tree search. It uses procedural knowledge about how individual players tend to achieve their goals in the domain, and employs this information to limit the search to the part of the game tree that is consistent with the players' goals. However, there are some limitations: goals are only abandoned if they are finished, policy which does not allow replacing the current plan with a potentially better one. Also, goals are assigned on unit-basis level and more units following the same goal require more computational effort than if they were grouped together under the same objective. This is more similar to a bottom-up approach, the goals in GB-GTS

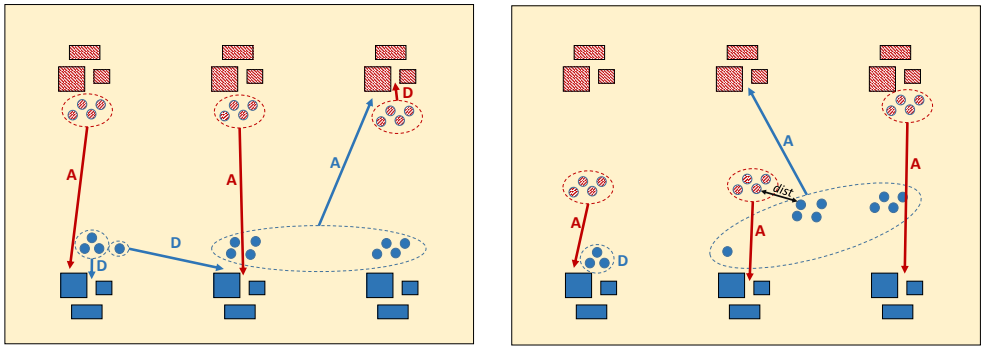
serving as building blocks for more complex behaviour. It contrasts with HTN-based approaches where the whole strategies are encoded using decompositions from the highest levels of abstraction to the lower ones.

6.3 Hierarchical Adversarial Search

The basic idea of our algorithm we call *Hierarchical Adversarial Search* is to decompose the decision making process into three layers: the first layer chooses a set of **objectives** that need to be accomplished to win the game (such as build an expansion, create an army, defend one of our bases or destroy an enemy base), the second layer generates action sequences to accomplish these objectives, and the third layer's task is to *execute* a plan. Here, executing means generating "real world" moves for individual units based on a given plan. Games can be played by applying these moves in succession. We can also use them to roll the world forward during look-ahead search which is crucial to evaluating our plans. An objective paired with a group of units to accomplish it is called a **task**. A pair of tasks (one for each player) constitutes a **partial game state**. We consider partial game states independent of each other. Hence, we ignore any interactions between units assigned to different partial game states. This assumption is the reason for the computational efficiency of the *hierarchical search* algorithm, because searching one state containing N units is much more expensive than searching M states with N/M units each. Lastly, a **plan** consists of disjoint partial game states, which combined represent the whole game state.

6.3.1 Application Domain and Motivating Example

Our implementation is based on SparCraft [38], which is an abstract combat simulator for StarCraft that is used in the current *UAlbertaBot* StarCraft AI to



(a) Initial plan (b) Re-planning

Figure 6.1: An example of two consecutive plans

estimate the chance of winning fights before engaging the enemy. Since simulation must be faster than real-time, SparCraft abstracts away game details such as building new structures, gathering resources or training units. Because of this limitation we consider simpler base-defense scenarios in this paper instead of playing full RTS games, and we focus on the lower two levels of the search: constructing and executing plans, while fixing the objective to destroying or defending bases. With extra implementation effort these scenarios could be set up in StarCraft as well, likely leading to similar results.

Fig. 6.1(a) shows a sample plan with three partial game states. The blue (solid, bottom) player chose three units to defend the left base (task 1), one unit to defend the middle base (task 2) and eight units to attack the enemy base to the right (task 3). Analogously, the red (striped, top) player chose two attacking tasks (tasks 1 and 2) and one defending task (task 3), assigning four units to each. In this example the first partial game state contains the first task of each player, the second partial game state the second task, and the last partial game state the third task. After a while the red units of the middle (second) partial game state get within attack range of the blue units assigned to the third partial game state and re-planning is triggered. One possible resulting scenario is shown in Fig. 6.1(b):

- the blue player still chooses to defend with three units against four on the left side in the first partial game state
- he switches his nine units to attack the middle enemy base, and is opposed by four enemy units in the second partial game state
- the blue player chooses to do nothing about the right base, so the third partial game state will only contain four enemy units attacking (blue player has an idle objective).

6.3.2 Bottom Layer: Plan Evaluation and Execution

The bottom layer serves two purposes: in the hierarchical search phase (1) it finds lowest-level actions sequences and rolls the world forward executing them, and in the plan execution phase (2) it will generate moves in the actual game. In both phases we first create a new sub-game that contains only the units in the specific partial game state. Then, in phase (1) we use fast players with scripted behaviour specialized for each objective to play the game, as the numbers of playouts will be big. During (1) we are not concerned with returning moves, but only with evaluating the plans generated by the middle layer. During phase (2) we use either Alpha-Beta or Portfolio search [43] to generate moves during each game frame, as shown in Fig. 6.2. The choice between Alpha-Beta or Portfolio search depends on the number of units in the partial game state. Portfolio search is currently the strongest algorithm for states with large numbers of units, while Alpha-Beta search is the strongest for small unit counts.

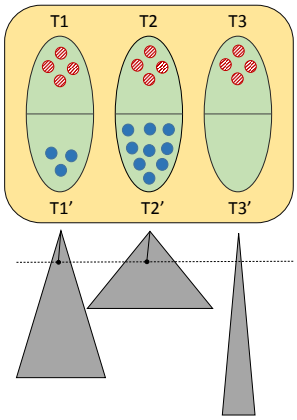


Figure 6.2: Alpha-Beta search on partial game states

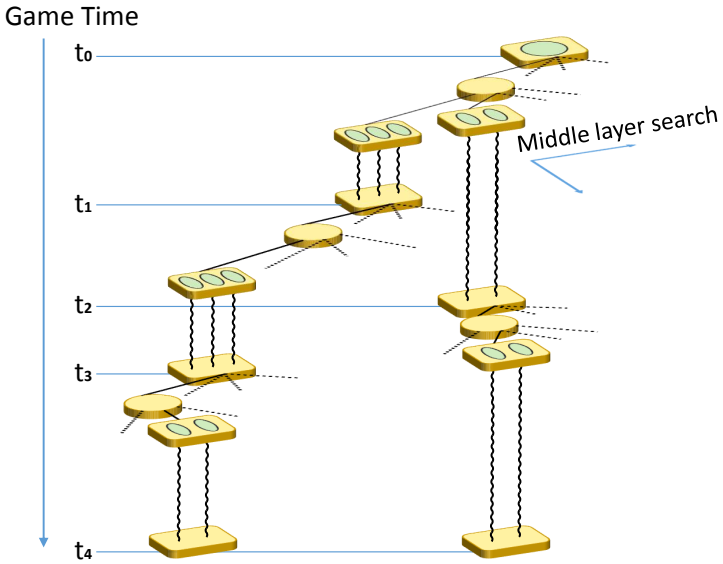


Figure 6.3: Middle layer search: the middle layer performs minimax search on abstract moves (rectangles depict max nodes, circles depict min nodes). The bottom layer executes scripted playouts to advance the game. Given more time, scripts can be replaced by search.

These processes are repeated in both phases until either a predefined time period has passed, or any task in the partial game state has been completed or is impossible to accomplish.

6.3.3 Middle Layer: Plan Selection

The middle layer search, shown in Fig. 6.3, explores possible plans and selects the best one for execution. The best plan is found by performing a minimax search (described in Algorithm 6.1), where each move consists of selecting a number of tasks that will accomplish some of the original objectives. As the number of moves (i.e., task combinations) can be big, we control the branching factor with a parameter X: we only consider X heuristically best moves for each player. With good move ordering we expect the impact in plan quality to be small. We sort the moves according to the average completion probability of the tasks. For each task, this probability is computed using the LTD2 ("Life-Time Damage 2") score (the sum of the square root of hit points remaining of each unit times its average attack value per frame [43]) of the player units assigned to that task compared to the LTD2 score of the enemy units closest to the task. We assume that enemy units influence only the task closest to them.

One of the players makes a move by generating some tasks (line 7) and applying them successively (line 9), the other player independently makes his

Algorithm 6.1. Hierarchical Adversarial Search (2 layers)

```
 1: procedure PlanSearch(Depth d, State s, Task oppTask, Player p)
 2:     best ← −∞;
 3:     if empty(oppTask) then                          ▷ First player gen. tasks
 4:         if endSearch() then
 5:             return evaluate(s)                      ▷ w.r.t. objectives
 6:         else
 7:             tasks ← genTasks(s, p)
 8:             for Task t in tasks do
 9:                 val ← - PlanSearch(d + 1,s,t,opp(p))
10:                 if val > best then
11:                     best ← val;
12:                 end if
13:             end for
14:             return best
15:         end if
16:     else                                            ▷ Second player gen. tasks
17:         tasks ← genTasks(s,p)
18:         plans ← genPlans(s,enemyTask,tasks,p)
19:         for Plan plan in plans do
20:             playout(plan)
21:             merge(s)                                ▷ Merge partial game states
22:             val ← - PlanSearch(d + 1,s,<>,opp(p))
23:             if value > best then
24:                 best ← val;
25:                 updatePrincipalVariation()
26:             end if
27:         end for
28:         return best
29:     end if
30: end procedure
```

move (line 17), and the opposing tasks are paired and combined into a plan consisting of several partial game states (line 18). We call a plan **consistent** if the partial game states do not interfere with each other (i.e., units from different partial game states cannot attack each other), and one example is shown in Fig. 6.1(b). If a plan is not *consistent* we skip it and generate the next plan to avoid returning a plan that would trigger an immediate re-plan. Otherwise, we do a playout using the scripted players for each partial game state and roll the world forward, until either one of the tasks is completed or impossible to accomplish or we reach a predefined game time (line 20). Fig. 6.3 shows two playouts at the same search depth, finishing at different times t_1 and t_2.

At this point we combine all partial game states into a full game state (line 21) and recursively continue the search (line 22). The maximum search depth is specified in game time, not search plies. Hence, some branches in which the playouts are cut short because of impossible or completed objectives might trigger more middle layer searches than others, like the leftmost branch in Fig. 6.3.

Finally, for leaf evaluation (line 5), we check the completion level of the top layer objectives (for example, how many enemy buildings we destroyed if we had a destroy base objective). Node evaluations can only be done at even depths, as we need both players to select a set of tasks before splitting the game into several partial game states.

After the middle layer completes the search, we will have a plan to execute: our moves from the principal variation of the minimax tree (line 25). At every game frame we update the partial game states in the plan with the unit data from the actual game, and we check if the plan is still valid. A plan might become invalid because one of its objectives is completed or impossible, or because units from two different partial game states are within attack range of each other. If the plan is invalid, re-planning is triggered. Otherwise it will be executed by the bottom layer using Alpha-Beta or Portfolio search for each partial game state, and a re-plan will be triggered when it is completed. To avoid re-planning very often, we do not proactively check if the enemy follows the principal variaton but follow the more lazy approach of re-planning if his actions interfere with our assumption of what his tasks are.

We did not implement the top layer and we simply consider destroy and defend all bases as objectives to be accomplished by the middle layer, but we discuss more alternatives in the last section, as future work.

6.3.4 Implementation Details

After receiving a number of objectives that should be completed from the top layer, the middle layer search has to choose a subset of objectives to be accomplished next. To generate a move we need to create a task for each objective in this subset: each task consists of some units assigned to accomplish a particular objective. Such assignments can be generated in various ways, for instance by spatial clustering. Our approach is to use a greedy bidding process similar to the one described in [177] that assigns units to tasks dependent on proximity and task success evaluations. The first step is to construct a matrix with bids for each of our units on each possible objective. Bids take into account the distance to the target and an estimate of the damage the unit will inflict and receive from enemy units close to the target. Let O be the total number of objectives, and $N \in \{1, \cdots, O\}$ a smaller number of objectives we want to carry out in this plan. We consider all permutations of N objectives out of O possible objectives (for a total of $O!/(O-N)!$). For example, if $O = 4$ and $N = 2$, we consider combinations $(1,2),(2,1),(1,3),(3,1),\ldots,(3,4),(4,3)$. The difference between (x,y) and (y,x) is objective priority: we assign units to accomplish objective x or y first.

For each of these combinations and each objective, we iterate assigning the unit with the highest bid to it until we think that the objective can be accomplished with a certain probability (we use 0.8 in the experiments) and then continue to the next objective. This probability is estimated using a

sigmoid function and the LTD2 scores of the Hierarchical Adversarial Search player's units assigned to complete that objective versus the LTD2 score of the enemy units closest to the objective. The moves will be sorted according to the average completion probability of all the O tasks (the assigned N as well as the $O - N$ we ignored). Consider the case in which one of the left-out objectives is a defend base objective. Consequently, the middle layer did not assign any units for this particular objective. In one hypothetical case there are no enemy units close to that particular base, so we will likely be able to accomplish it nonetheless. In another case, that particular base is under attack, so there will be a very low probability of saving it as there are no defender units assigned for this task. If the middle layer sorts the moves only using the probabilities for the N tasks to which it has assigned units, it will not be able to differentiate between the previous two cases and will treat them as equally desirable. Hence, we need to consider the completion probability of all tasks in the move score, even those to which the middle layer did not assign units.

After both players choose a set of tasks, we need to match these tasks to construct partial game states. Matching is done based on distances between different tasks (currently defined as the minimum distance between two units assigned to those tasks). Because partial game states are considered independent, we assign tasks that are close to each other to the same partial game state. For example, "destroy base A" for player 1 can be matched with "defend base A" for player 2 if the attacking units are close to base A. If the units ordered to destroy base A for player 1 are close to the units that have to destroy base B for player 2, then we have to match them because the involved units will probably get into a fight in the near future. However, we now also need to add the units from "defend base A" and "defend base B" to the same partial game state (if they exist) to avoid conflicts. After matching, if there are any unmatched tasks, we add dummy idle tasks for the opponent.

While performing minimax search, some of the plans generated by task matching might still be illegal (i.e., units in one partial game state are too close to enemy units from another partial game state). In that case we skip the faulty plan and generate the next one.

In our current implementation, when re-planning is needed, we take the time needed to search for a new plan. In an actual bot playing in a tournament, we would have a current plan that is being executed, and a background thread searching for a new plan. Whenever the current plan is deemed illegal, we would switch to the latest plan found. If that plan is illegal, or if the search for a new plan has not finished, we would fall back to another algorithm, such as Alpha-Beta, UCT, or Portfolio Search.

For plan execution, the time available at each frame is divided between different Alpha-Beta searches for each partial game state. The allotted time is proportional to the number of units in each, which is highly correlated to the branching factor of the search. As the searches usually require exponential ef-

fort rather than linear, more complex time division rules should be investigated in the future.

It is worth noting that both plan execution and evaluation are easily parallelizable, because each partial game state is independent of each other. Plan search can be parallelized with any minimax parallelization technique such as ABDADA [244]. Because at each node in the plan search we only explore a fixed number of children and ignore the others, the search might also be amenable to random sampling search methods, such as Monte Carlo tree search.

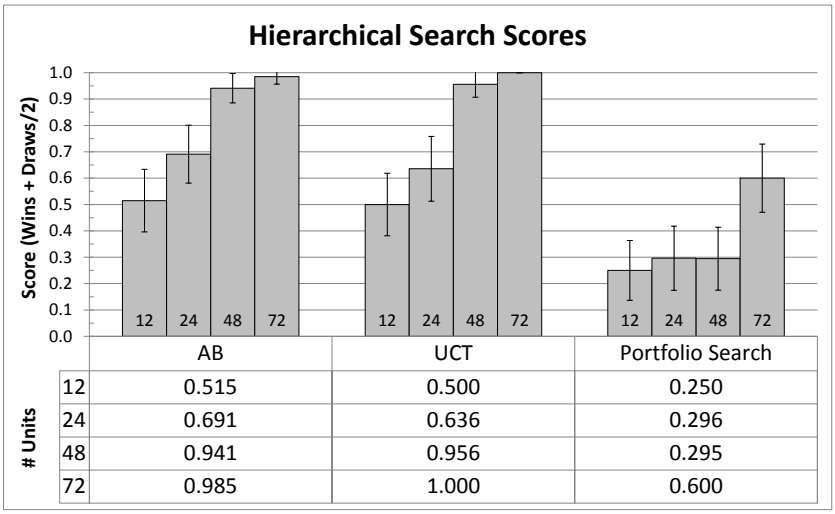
6.4 Empirical Evaluation

Two sets of experiments were carried out to evaluate the performance of the new Hierarchical Adversarial Search algorithm. In the first set we let the Hierarchical Adversarial Search agent which uses Portfolio Search for plan execution play against the Alpha-Beta, UCT, and Portfolio Search agents presented in [43]. All agents have the same amount of time available to generate a move. In the second set of experiments Hierarchical Adversarial Search using Alpha-Beta Search for plan execution plays against an Alpha-Beta Search agent whose allocated time varies.

Similar to [43], each experiment consists of a series of combat scenarios in which each player controls an identical group of StarCraft units and three bases. Each base consists of four (on half of the maps) or five (on the other half) structures: some are buildings that cannot attack but can be attacked (which are the objectives in our game), and some are static defences that can attack (e.g., photon cannons in StarCraft). Both players play the Protoss race and the initial unit/building placement is symmetric to ensure fairness.

The battlefields are empty StarCraft maps of medium size (128×72 tiles) without obstacles or plateaus because none of the tested algorithms has access to a pathfinding module. Experiments are done on four different maps, with three bases totalling 12 to 15 buildings for each player. The set-up is similar to Fig. 6.1(a), though on some maps the position of the middle base for the two players is swapped. To investigate the performance of the different algorithms we vary the number of units — 12, 24, 48, or 72 per player, equally split between the three bases.

For the first experiment we generate 60 battles for each unit configuration. For the second experiment we use only one map with 72 units per player, and play 10 games for each Alpha-Beta time limit. Each algorithm was given a 40 ms time limit per search episode to return a move. This time limit was chosen to comply to real-time performance restrictions in StarCraft, which runs at 24 frames per second (42 ms per frame). The versions of Alpha-Beta, UCT, and Portfolio Search (depending on parameters such as maximum limit of children per search node, transposition table size, exploration constant or evaluation) are identical to those used in [43]. It is important to note that the Alpha-Beta



# Units	AB	UCT	Portfolio Search
12	0.515	0.500	0.250
24	0.691	0.636	0.296
48	0.941	0.956	0.295
72	0.985	1.000	0.600

Figure 6.4: Results of Hierarchical Adversarial Search against Alpha-Beta, UCT and Portfolio Search for combat scenarios with n vs. n units ($n = 12, 24, 48$ and 72). 60 scenarios were played allowing 40 ms for each move. 95% confidence intervals are shown for each experiment.

Search agent uses a fall-back strategy: in cases when it cannot complete even a 1-ply search it will execute scripted moves. In our experiments we impose a time limit of 3000 frames for a game to finish, at which time we decide the winner using the LTD2 score. All experiments were performed on an Intel(R) Core2 Duo P8400 CPU 2.26GHz with 4 GB RAM running Fedora 20, using single-thread implementations. The software was implemented in C++ and compiled with g++ 4.8.2 using -O3 optimization.

6.4.1 Results

In the first set of experiments, we compare Hierarchical Adversarial Search using Portfolio Search for partial game states against Alpha-Beta, UCT and Portfolio Search. Fig. 6.4 shows the win rate of the hierarchical player against the benchmark agents, grouped by number of units controlled by each player. Hierarchical Adversarial Search has similar performances against Alpha-Beta and UCT algorithms. It is roughly equal in strength when playing games with the smallest number of units, but its performance grows when increasing the number of units. From a tie at 12 vs. 12 units, our algorithm exceeds 90% winning ratio for 48 vs. 48 units and reaches 98% for the largest scenario. As expected, performing tree searches on the smaller game states is more efficient than searching the full game state, as the number of units grows. Against Portfolio Search, our method does worse on the smaller scenarios. Hierarchical Adversarial Search overcomes Portfolio Search only in the 72 vs. 72 games, which are too complex for Portfolio to execute the full search in the allocated 40 ms.

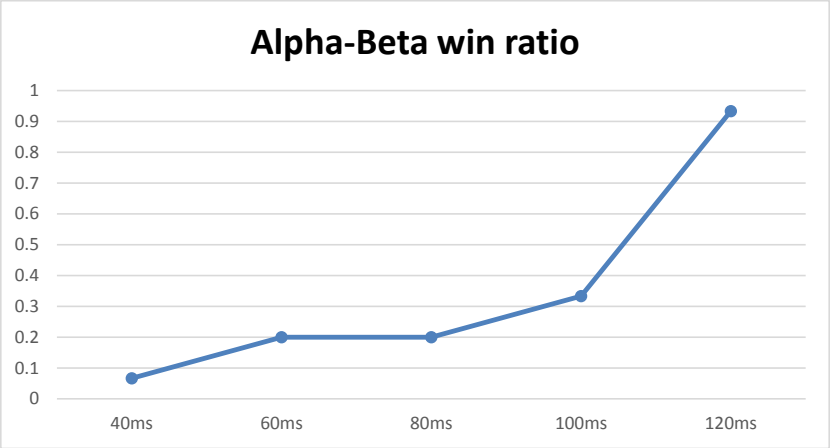


Figure 6.5: Alpha-Beta Search strength variation when increasing computation time, playing against Hierarchical Adversarial Search using 40 ms per move and Alpha-Beta Search for plan execution. 10 experiments using 72 units on each side were performed for each configuration.

In the second set of experiments, we compare Hierarchical Adversarial Search using Alpha-Beta Search for partial game states against Alpha-Beta Search. Hierarchical Adversarial Search computing time is limited to 40 ms and Alpha-Beta Search agent's time was varied using 20 ms steps to find at which point the advantage gained by partitioning the game state can be matched by more computation time. Fig. 6.5 shows the win ratio of Alpha-Beta in scenarios with 72 units for each player. Hierarchical Adversarial Search still wins when Alpha-Beta Search has double the search time (80 ms), but loses when tripling Alpha-Beta's time. These results suggest that our Hierarchical Abstract Search implementation can be improved by complete enumeration of tasks in the middle layer and/or conducting searches in the bottom layer once unit counts drop below a certain threshold.

6.5 Conclusions and Future Work

In this paper we have presented a framework that attempts to employ search methods for different abstraction levels in a real-time strategy game. In large scale combat experiments using SparCraft, our Hierarchical Adversarial Search algorithm outperforms adaptations of Alpha-Beta and UCT Search for games with simultaneous and durative moves, as well as state-of-the-art Portfolio search. The major improvement over Portfolio Search is that Hierarchical Adversarial Search can potentially encompass most areas of RTS game playing, such as building expansions and training armies, as well as combat.

As for future work, the first area of enhancements we envision for our work is adding pathfinding to Alpha-Beta, UCT, and Portfolio Search to test Hier-

archical Adversarial Search on real StarCraft maps. We also plan to improve the task bidding process by using a machine learning approach similar to the one described in [206] for predicting battle outcomes. It would still be fast enough but possibly much more accurate than using LTD2 scores, and could also integrate a form of opponent modelling into our algorithm. Finally, we currently only use two of the three layers and generate fixed objectives for the middle layer (such as destroy opponent bases) instead of using another search algorithm at the top layer. If we extend SparCraft to model StarCraft's economy, allowing the agents to gather resources, construct buildings, and train new units, the top layer decisions become more complex and we may have to increase the number of abstraction layers to find plans that strike a balance between strengthening the economy, building an army, and finally engaging in combat.

6.6 Contributions Breakdown and Updates Since Publication

My main contributions to this work were focused on the spatial decomposition of the state and the bottom layer implementation, while the second author helped with the top and middle layer architectures. We collaborated on the overall design with Michael Buro, who also supervised the work.

An alternative abstract search mechanism was presented [232, 233] at the same conference as the research presented in this chapter. While our algorithm is using the lowest level to forward the world, the authors abstract the game state and perform look-ahead search at this abstract level directly. No experiments were ever performed to assess the relative strengths and weaknesses of each approach.

Hierarchical task decomposition combined with minimax search was also the core idea in other similar research [164]. However, while we used three fixed, separate levels of abstractions the authors relied on scripted actions in the form of HTN planning.

Other research that uses configurable scripts as an action abstraction mechanism is presented in Chapter 8. This algorithm, called Puppet Search, outperformed the HTN based approach mentioned above. Despite promising results, we chose to discontinue the method presented in this chapter in favor of Puppet Search because of the difficulties in hand-crafting the multiple search layers and abstraction levels. Puppet Search offers a more elegant way of implementing a similar concept – adversarial search with state and action abstraction – by using already common action scripts as the first abstraction level.

Chapter 7

Evaluating Real-Time Strategy Game States Using Convolutional Neural Networks

Abstract

Real-time strategy (RTS) games, such as Blizzard's StarCraft, are fast paced war simulation games in which players have to manage economies, control many dozens of units, and deal with uncertainty about opposing unit locations in real-time. Even in perfect information settings, constructing strong AI systems has been difficult due to enormous state and action spaces and the lack of good state evaluation functions and high-level action abstractions. To this day, good human players are still handily defeating the best RTS game AI systems, but this may change in the near future given the recent success of deep convolutional neural networks (CNNs) in computer Go, which demonstrated how networks can be used for evaluating complex game states accurately and to focus look-ahead search.

In this paper we present a CNN for RTS game state evaluation that goes beyond commonly used material based evaluations by also taking spatial relations between units into account. We evaluate the CNN's performance by comparing it with various other evaluation functions by means of tournaments played by several state-of-the-art search algorithms. We find that, despite its much slower evaluation speed, on average the CNN based search performs significantly better compared to simpler but faster evaluations. These promising initial results together with recent advances in hierarchical search suggest that dominating human players in RTS games may not be far off.

7.1 Introduction

The recent success of AlphaGo [195], culminating in the 4-1 win against one of the strongest human Go players, illustrated the effectiveness of combining Monte Carlo Tree Search (MCTS) and deep learning techniques. For AlphaGo, convolutional neural networks (CNNs) [134, 127] were trained to mitigate the prohibitively large search space of the game of Go in two ways: First, a policy network was trained, using both supervised and reinforcement learning techniques, to return a probability distribution over all possible moves, thereby focusing the search on the most promising branches. Second, MCTS state evaluation accuracy was improved by using both value network evaluations and playout results.

In two-player, zero-sum games, such as Chess and Go, optimal moves can be computed by using the minimax rule that minimizes worst case loss. In theory, these games can be solved by recursively applying this rule until reaching terminal states. However, in practice completely searching the game tree is infeasible, the procedure must be cut short, and an approximate evaluation function must be used to estimate the value of the game state. Because states closer to the end of the game are typically evaluated more accurately, deeper search produces better moves. But as game playing agents often have to make their move decision under demanding time constraints, great performance gains can be achieved by improving the evaluation function's accuracy.

The size of the state space of the game of Go, although much larger than that of Chess, is tiny in comparison to real-time strategy (RTS) games such as Blizzard's StarCraft. In the game of Go, at every turn, a single stone can be placed at any valid location on the 19×19 board and the average game length is around 150 moves. In RTS games, each player can simultaneously command many units to perform a large number of possible actions. Also, a single game can last for tens of thousands of simulation frames, with possibly multiple moves being issued in each one. Moreover, RTS game maps are generally much larger than Go boards and feature terrain that often affects movement, combat, and resource gathering. Therefore, for RTS games, good state evaluations and search control, such as using policy networks, plays an even greater role.

CNNs are adept at learning complex relationships within structured data due to their ability to learn hierarchies of abstract, localized representations in an end-to-end manner [127]. In this paper we investigate the effectiveness of training a CNN to learn the value of game states for a simple RTS game and show significant improvement in accuracy over simpler state-of-the-art evaluations. We also show that incorporating the resulting learned evaluation function into state-of-the-art RTS search algorithms increases agent playing strength considerably.

7.2 Related Work

Search based planning approaches have had a long tradition in the construction of strong AI agents for abstract games like Chess and Go, and in recent years they have progressively been applied to modern video games, especially the RTS game StarCraft. This is a difficult endeavor due to the enormous state and action spaces, and finding optimal moves under tight real-time constraints is infeasible for all but the smallest scenarios. Consequently, the research focus in this area has been on reducing the search space via different abstraction mechanisms and on producing good state evaluation functions to guide this search effort.

In this section we briefly discuss some of these attempts, starting with various methods used for state evaluation in RTS games. We then present recent research on deep neural networks and their use in game playing agents.

7.2.1 State Evaluation in RTS Games

Playing RTS games well requires strategic as well as tactical skills, ranging from building effective economies, over deciding what to build next based on scouting results, to maneuvering units in combat encounters. In RTS game combat each player controls an army consisting of different types of units and tries to defeat the opponent's army while minimizing its own losses. Because battles have a big impact on the result of RTS games, predicting their outcome accurately is very important, especially for look-ahead search algorithms.

A common metric for estimating combat outcomes is LTD2 [126], which is based on the lifetime damage each unit can inflict. LTD2 was used, in conjunction with short deterministic playouts, for node evaluation in alpha-beta search to select combat orders for individual units [44]. A similar metric was later used as state evaluation, this time combined with randomized playouts [161, 164].

Likewise, Hierarchical Adversarial Search [200] requires estimates of combat outcomes for state evaluation and uses a simulator for this purpose. However, because simulations become more expensive as the number of units grows, faster prediction methods are needed. For instance, a probabilistic graphical model trained on simulated battles can accurately predict the winner [206]. This model, however, has several limitations such as not modeling damaged units and not distinguishing between melee and ranged combat. Another model, based on Lanchester's attrition laws [202], does not have such shortcomings. It takes into account the relative strength of different unit types, their health and the fact that ranged weapons enable units to engage several targets without having to move, which causes a non-linear relationship between army size differences and winning potential. After learning unit strength values offline using maximum likelihood estimation from past recorded battles, this improved model has been successfully used for state evaluation in a state-

of-the-art RTS search algorithm [11].

All mentioned approaches focus on a single strategic component of RTS games, (i.e., combat), and lack spatial reasoning abilities, ignoring information such as unit positions and terrain. Global state evaluation in complex RTS games such as StarCraft has been less successful [59], likely due to the limited expressiveness of the linear model used.

7.2.2 Neural Networks

In recent years deep convolutional neural networks (CNNs) have sparked a revolution in AI. The spectacular results achieved in image classification [127] have led to deep CNNs being effectively applied to a wide range of domains. For vision tasks, CNNs have been applied to object localization [77], segmentation [141], facial recognition [189], super-resolution [53] and camera-localization [120] to name just a few examples, all the while continuing to make further progress in image classification [222]. Deep CNNs have also been successfully applied to tasks as diverse as natural language categorization [114, 255], translation [7] and algorithm learning [119].

Deep CNNs owe their success to their ability to learn multiple levels of abstraction, each one building upon abstractions learned in previous layers. More specifically, deep CNNs learn a hierarchy of spatially invariant, localized representations, each layer aggregating and building upon representations in previous layers toward the combined goal of minimizing loss [77].

There is a long history of using simple linear regression and shallow neural networks to construct strong AI systems for classic board games such as Backgammon and Othello [69]. However, scaling up state evaluations to more complex games such as Go only became possible when it was discovered how to effectively train weights in deep neural networks, which can be considerably more expressive than shallow networks with the same number of weights [135].

Since then CNNs have been successfully used to play Atari video games with a policy network trained by supervised learning, using training data generated by a slow but strong UCT player [85]. Similar networks have been trained with reinforcement learning [149, 150]. Most remarkable, however, is the recent 4-1 win of AlphaGo [195], a deep CNN based Go playing program, over one of today's best Go players Lee Sedol. AlphaGo combines MCTS with deep CNNs for state evaluation and move selection that were trained by supervised and reinforcement learning.

This historic accomplishment sparks hope that CNNs can also be used for even more complex tasks, such as playing real-time games with imperfect information – a domain still dominated by human players.

7.3 A Neural Network for RTS Game State Evaluation

In this section we describe the dataset, the neural network structure and the procedure used for training a state evaluation network for μRTS[1], a simple RTS game designed for testing AI techniques. μRTS provides the essential features of an RTS game: it supports four unit and two building types, all of them occupying one tile, and there is only one resource type. The game state is fully observable. μRTS supports configurable map sizes, commonly ranging from 8×8 to 16×16 in published papers. The game user interface and details about the unit types are shown in Figure 7.1. μRTS comes with a few basic scripted players, as well as search based players implementing several state-of-the-art RTS search techniques [161, 194, 164], making it an useful tool for benchmarking new AI algorithms.

The purpose of the neural network we describe here is to approximate the value function $v^*(s)$, which represents the win-draw-loss outcome of the game starting in state s assuming perfect play on both sides.

In practice we have to approximate this value function with v_θ, for instance by using a neural network with weights θ. These weights are trained by regression on state-outcome pairs (s, w), using stochastic gradient descent to minimize the mean squared error between the predicted value $v_\theta(s)$ and the corresponding outcome w. The output of our network will be the players' probabilities of winning the game when starting from the input position.

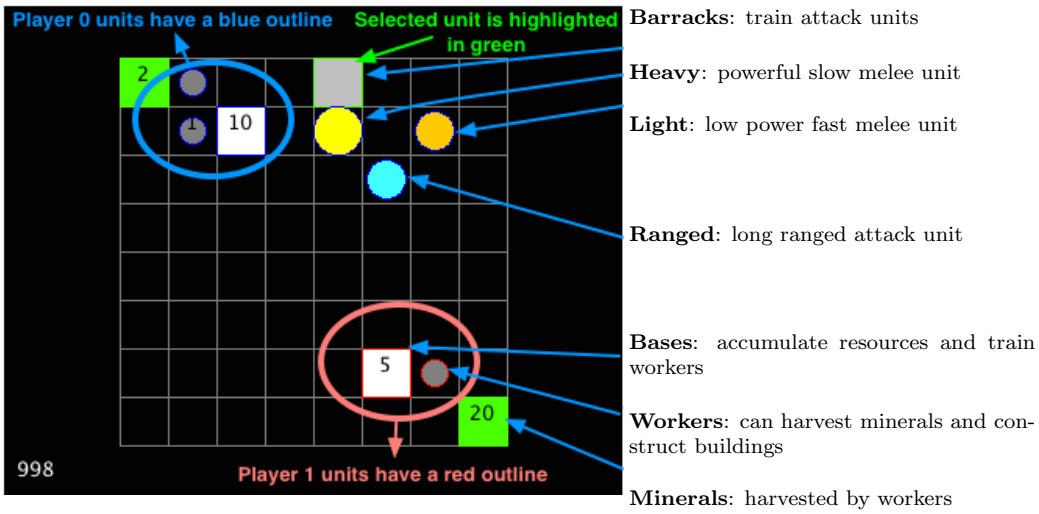


Figure 7.1: Screenshot of μRTS, with explanations of the different in-game symbols.

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

7.3.1 Data

The dataset used for training the neural network was created by playing round-robin tournaments between 15 different μRTS bots, 11 of which are included in the default μRTS implementation. The other 4 are versions of the Puppet Search algorithm [11]. Each tournament consists of $(15 \times 14)/2 = 105$ matches. One 8×8 map was used, with 24 different initial starting conditions. All scenarios start with one base and one worker for each player, but with different, symmetric, initial positions. These tournaments were played under four different time limits: maximums of $100ms$, $200ms$, 100 playouts and 200 playouts per search episode. In total $105 \times 24 \times 4 = 10\,080$ different games were played from which draws were discarded ($\approx 8\%$).

Predicting game outcomes from data consisting of complete games leads to overfitting because while successive states are strongly correlated, the regression target is shared for the entire game. To mitigate the problem, the authors of AlphaGo [195] add only a single training example (s, w) to the dataset from each game. Because we have significantly less data (10 thousand vs. 30 million episodes), we chose to sample 3 random positions from each game. As a result, for game i we add $\{(s_{i1}, w_i), (s_{i2}, w_i), (s_{i3}, w_i)\}$ to the dataset, and slightly over 25 000 positions are generated.

The dataset was split into a test set (5 000 positions) and a training set (the remaining 20 000 positions). Finally, the training set was augmented by including all reflections and rotations of each position for a total of 160 000 positions.

7.3.2 Features

Each position s is preprocessed into a set of 8×8 feature planes. These features correspond to the raw board representation and contain information about each tile of the μRTS map: unit ownership and type, current health points, game frames until actions are completed and resources.

All integers, such as unit health points, are split into K different 8×8 planes of binary values using the one-hot encoding. For example, five separate binary feature planes are used to represent whether an unit has $1, 2, 3, 4$ or ≥ 5 health points. The full set of feature planes is listed in Table 7.1.

7.3.3 Network Architecture & Training Details

The input to the neural network is an 8×8×25 image stack consisting of 25 feature planes. There are two convolutional layers that pad the input with zeros to obtain a 10×10 image. Each then is convolved with 64 and respectively 32 filters of size 3×3 with stride 1. Both are followed by leaky rectified linear units (LReLUs) [251, 100]. A third hidden layer convolves 1 filter of size 1×1 with stride 1, again followed by an LReLU. Then follow two fully connected (dense) linear layers, with 128 and 64 LReLU units, respectively. A dropout

Table 7.1: Input feature planes for the neural network.

Feature	# of planes	Description
Unit type	6	Base, Barracks, worker, light, ranged, heavy
Unit health	5	$1, 2, 3, 4, \text{or} \geq 5$
Unit owner	2	Masks to indicate all units belonging to one player
Frames to completion	5	$0-25$, $26-50$, $51-80$, $81-120$, or ≥ 121
Resources	7	1, 2, 3, 4, 5, $6-9$, or ≥ 10

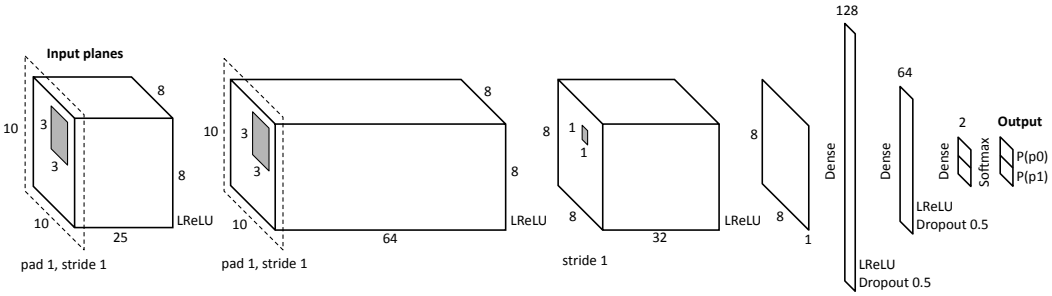


Figure 7.2: Neural network architecture.

ratio of 0.5 is applied to both fully connected layers. The output layer is a fully connected layer with two units, and a softmax function is applied to obtain the winning probabilities for player 0 and player 1 ($P(p0)$ and $P(p1)$). All LReLUs have negative slope of $\alpha = -1/5.5$. The resulting architecture is shown in Figure 7.2.

Our architecture was motivated by current trends toward the use of small filter sizes ($\leq 3\times3$), few (or no) pooling layers, and same-padded convolution (multiple layers of the same width and height, each layer padded with zeros following convolution) [99, 195]. We were also guided by the principle of gradually decreasing the dimension of internal representations as one moves from input toward task; one example being the reduction from 64 to 32 filters, another being the use of 1×1 convolutions for dimensionality reduction [222]. This principle can also be seen in the fully connected layers. LReLUs were used following suggestions from [251] and [100].

Before training, we used Xavier random weight initialization [79] which equalizes signal variance. During training, the stepsize alpha was initialized to 0.00001 and was multiplied by 0.2 every $100K$ training steps. We used adaptive moment estimation (ADAM) with default values of $\beta_1 = 0.9, \beta_2 = 0.999, \varepsilon = 10^{-8}$ as suggested in [121]. The network was trained for $400K$ mini-batches of 64 positions, a process which took approximately 20 minutes on a single GPU to converge.

For training, we used the Python (2.7.6) interface to Caffe [113], utilizing CUDA[2] version 7.5 and cuDNN[3] version 4. The machine used for training the neural network had an Intel(R) Pentium(R) CPU G2120 3.10GHz processor, 8 GB RAM and one GeForce GTX 760 GPU (1152 cores and 4 GB memory) running Linux Mint 17.3.

7.4 Experiments and Results

All experiments that are reported below were performed on Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz with 8 GB RAM machines running Ubuntu 14.04. The test machines do not have CUDA capability 3 and the neural network computations were run solely on the CPU. μRTS software is implemented in Java and compiled and run with JDK 8u74.

7.4.1 Winner Prediction Accuracy

In the first set of experiments we compare the speed and accuracy of our neural network for evaluating game states with a Lanchester model [202] and a simple evaluation function that takes into account the cost and health points of units and the resources each player has. This is the default evaluation function that the μRTS search algorithms use. In equation 7.1 player indices are either 0 or 1: $player \in \{0, 1\}$.

$$eval(player) = E_{player} - E_{1-player} \tag{7.1}$$

In equation 7.2, R_p is the amount of resources a player currently has, W_p is a player's set of workers, R_u is the amount of resources each worker unit is carrying, C_u is the cost of unit u, HP_u the current health points of unit u, and MaxHP_u its maximum health points. $W_{res}, W_{work}, W_{unit}$ are constant weights.

$$E_p = W_{res}R_p + W_{work}\sum_{u \in W_p} R_u + W_{unit}\sum_{u \in p}\frac{C_u \text{HP}_u}{\text{MaxHP}_u} \tag{7.2}$$

Two versions of this simple evaluation functions were used: one with μRTS's default weights, and one optimized via logistic regression on the same training set used for the neural network. The Lanchester model keeps the two resource terms of the simple evaluation function but revises the army's impact. While the contribution of the buildings is similar to equation 7.2, a new term is added for combat units:

$$E'_p = W_{res}R_p + W_{work}\sum_{u \in W_p} R_u + W_{base}\frac{\text{HP}_{base}}{\text{MaxHP}_{base}} + W_{barracks}\frac{\text{HP}_{barracks}}{\text{MaxHP}_{barracks}} + N_p^{(o-1)}\sum_{u \in p}\alpha_u\frac{\text{HP}_u}{\text{MaxHP}_u} \tag{7.3}$$

[2] `https://developer.nvidia.com/cuda-toolkit`

[3] `https://developer.nvidia.com/cudnn`

In equation 7.3, α_u is a strength value unique to each unit type, N_p is the total number of units of player p and o is the Lanchester attrition order. For μRTS, our experiments suggest that an attrition order of $o = 1.7$ works best on average if we had to choose a fixed order for all possible encounters. The four W and four α constants (one for each unit type) are optimized with logistic regression.

Figure 7.3 shows the position evaluation accuracy of the neural network, compared to the default μRTS evaluation function, the optimized version and the Lanchester model, on the previously described test set of 5 000 positions sampled from bot games. A scripted playout evaluation was also tested, in which the position is played until the end using the WorkerRush script, described in section 7.4.2, to generate moves for both players. Values of 1, 0 or -1 are returned, corresponding to a player 0 win, draw or loss, respectively. The WorkerRush script is the strongest of the four scripts described in the next section, and produces the most accurate winner prediction function, though slightly worse than the simple evaluation function. A random playout was also tried, but it performed even worse.

The neural network is consistently more accurate during the first half of the game. At the beginning of the game before any unit has been built, the simple evaluation function and the Lanchester model mostly predict draws, because they do not take positional information into account. During the second half

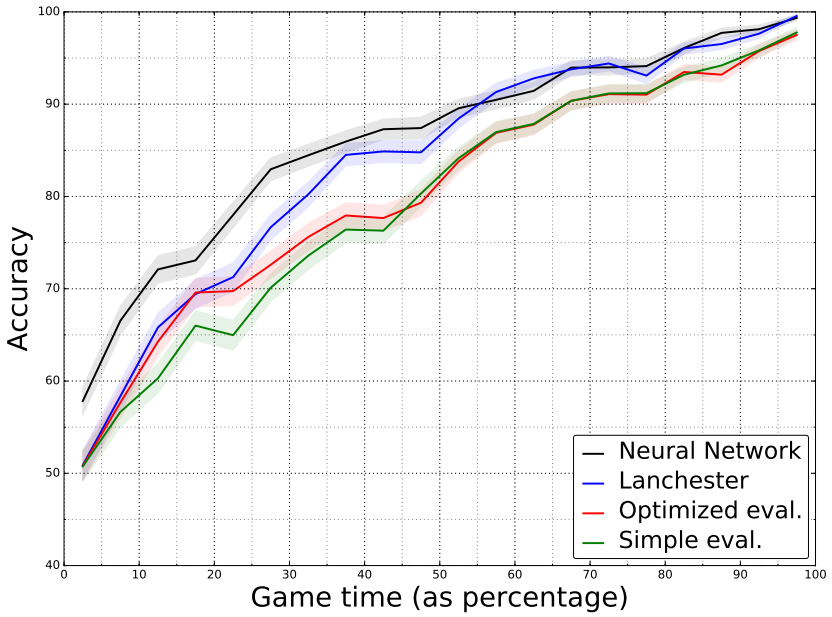


Figure 7.3: Comparison of evaluation accuracy between the neural network, μRTS's built-in evaluation function, its optimized version and the Lanchester model. The accuracy at predicting the winner of the game is plotted against the stage of the game, expressed as a percentage of the game length. Results are aggregated in 5% buckets. Shaded area represents one standard error.

of the game army balance is more relevant, and both the neural network and the Lanchester model perform better than the simple evaluation functions.

The average time needed for a single simple evaluation is $0.012\mu s$, the Lanchester model takes $0.087\mu s$, while a full network evaluation on the CPU takes $147\mu s$. This time includes processing the games state into feature planes, sending the data to a Python thread (on the same CPU core as the search algorithm), running a forward pass on the network and returning the outcome. The network evaluation takes close to two thirds of the time, around $102\mu s$. We tested the speed of the network evaluation on a GPU as well. On a mid-range NVIDIA GTX 760, the time is slightly shorter than the CPU-only version ($118\mu s$).

However, processing only one position at a time does not take advantage of the pipelined GPU architecture. To measure potential gains of evaluating positions in parallel, we ran batches of 256 positions whose evaluation took $10\,707\mu s$, of which $9\,985\mu s$ was spent on the CPU (feature planes) and $722\mu s$ on the GPU, for an average of $2.8\mu s$ of GPU time per evaluation. A search algorithm — like AlphaGo's — that can perform leaf evaluations asynchronously would benefit greatly from doing state evaluations on the GPU.

7.4.2 State Evaluation in Search Algorithms

A second set of experiments compares the performance of four game tree search algorithms — ϵ-Greedy MCTS, Naïve MCTS, AHTN-F and AHTN-P, described below — when using the simple evaluation function, the optimized evaluation function, the Lanchester model or the neural network for state evaluation.

The sixteen resulting algorithms played against the following eleven opponents provided by the μRTS implementation, all using default parameters and the simple μRTS evaluation function:

WorkerRush: a hardcoded rush strategy that constantly produces workers and sends them to attack.

LightRush: builds a barracks, and then constantly produces *light* military units to attack the nearest target (it uses one worker to mine resources).

RangedRush: is identical to **LightRush**, except for producing *ranged* units.

HeavyRush: is identical to **LightRush**, except for producing slower but stronger *heavy* units.

MonteCarlo(MC): a standard Monte Carlo search algorithm: for each legal player action, it runs as many simulations as possible to estimate their expected reward.

ϵ-Greedy MC: Monte Carlo search, but using an ϵ-greedy sampling strategy.

Naïve MCTS: Monte Carlo Tree Search algorithm with a sampling strategy specifically designed for games with combinatorial branching factors, such as RTS games. This strategy, called *Naïve Sampling*, exploits the particular tree structure of games that can be modeled as a Combinatorial Multi-Armed Bandit [161].

ϵ-Greedy MCTS: like NaïveMCTS, but using an ϵ-greedy sampling strategy.

MinMax Strategy: for a set of strategies (WorkerRush, LightRush, RangedRush and Random), playouts are run for all possible pairings. It approximates the Nash equilibrium strategy using the minimax rule, whereby one player (Max) maximizes its payoff value while the other player tries to minimize Max's payoff [180].

AHTN-P: an Adversarial Hierarchical Task Network, that combines minimax game tree search with HTN planning [164]. In this AHTN definition the main task of the game can be achieved only by three non-primitive tasks (abstract actions that decompose into actions that agents can directly execute in the game). The tasks are three rushes with three different unit types.

AHTN-F: a more elaborate AHTN with a larger number of non-primitive tasks for harvesting resources, training units of different types, or attacking the enemy.

All search based algorithms (bottom seven in the list above) evaluate states by running a short playout of 100 frames. The playouts are performed using a random policy in which non-move actions (harvest, attack, build) have a higher probability than moves. The only exception is MinMax, whose playouts are 400 frames long, because it only does 16 playouts — one for each pair of strategies — and uses its fixed set of strategies instead of the random policy. The resulting states are evaluated with the simple evaluation function in equation 7.1, the optimized function, the Lanchester model or the neural network.

Every player has a computational budget of either a given time duration or a maximum number of state evaluations per game frame. Moreover, players can split the search process over multiple frames; for example, if the game state does not change during 10 game frames before a player needs to issue an action, then players have ten times the budget to issue actions. We call this consolidated budget a *search episode*.

In the tournament each of the 176 matchups consists of 24 games played on an 8×8 map, with different but symmetric starting positions. To compute the score, every win is worth 1 point, and if the game reaches 3 000 frames, it is considered a draw, and awarded 0.5 points.

Figure 7.4 summarizes the average win rate against all opponents when using the different evaluation methods. On average, the neural network shows

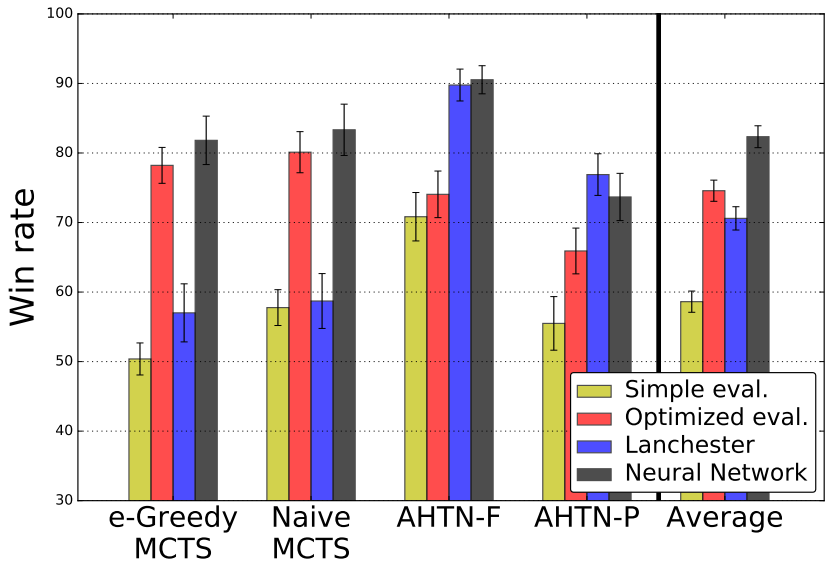


Figure 7.4: Average win rate against all opponents when using the simple evaluation function described in equations 7.1 and 7.2, the same function with optimized weights, the Lanchester model or the neural network described in section 7.3. Each algorithm has **200 milliseconds** of search time per frame. Error bars show one standard error.

over 10% higher win rates than the other methods. Moreover, the performance of the neural network is consistent across all four algorithms, while the results of the optimized evaluation and the Lanchester model fluctuate depending on the underlying search algorithm type.

Table 7.2 shows the average number of nodes expanded per search episode. The average length of a search episode in the tournament games was around seven frames. Slow search algorithms such as AHTNs are less affected by a slow state evaluation, as most of their computational effort is expended in the tree phase. As a result, the AHTNs perform better when using the most accurate functions, regardless of their speed. The balance on the faster

Table 7.2: Nodes expanded per search episode, when running with a maximum time limit of 200ms per frame.

	Average # nodes expanded per search episode		
AI Algorithm	Simple Evaluation	Lanchester	Neural Network
ϵ-Greedy MCTS	16834	14682	1069
Naïve MCTS	16654	13876	1122
AHTN-F	937	969	507
AHTN-P	134	125	123

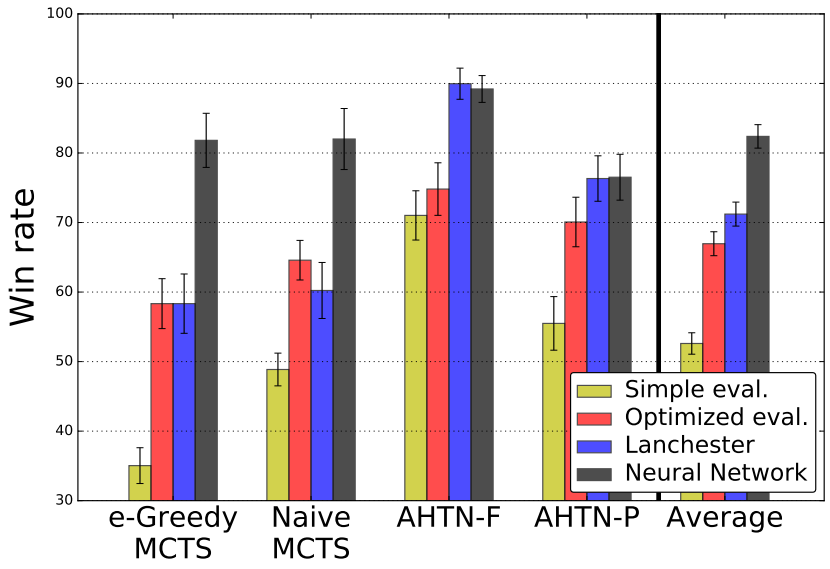


Figure 7.5: Average win rate against all opponents when using the simple evaluation function described in equations 7.1 and 7.2, the same function with optimized weights, the Lanchester model or the neural network described in section 7.3. Each algorithm is allowed to expand **200 nodes** per frame. Error bars show one standard error.

MCTS algorithms is more delicate, with both the fast optimized evaluation function and the neural network outperforming the relatively accurate and fast Lanchester model. The ∼1% accuracy increase in the first quarter of the game between the optimized simple evaluation and Lanchester is not enough to offset the ∼15% less nodes per second. However, the ∼7% accuracy gain of the neural network more than makes up for its ∼93% speed loss. Improving the accuracy of the evaluation function at the beginning of the game is important, as early game decisions likely have a large impact on the game outcome.

Figure 7.5 shows a summary of a similar tournament using a limit of 200 state evaluations per frame, rather than 200 milliseconds. The fastest simple evaluation function shows significantly worse performance on the MCTS algorithms, because in this experiment only the evaluation accuracy is relevant, not the speed.

To scale the neural network to larger map sizes and more complex games, the size of the network will likely have to increase, both in the size of each layer and in the number of layers. This expansion will lead to slower evaluation times. However, we have shown that a small increase in evaluation accuracy is able to compensate for several orders of magnitude in speed reduction. Furthermore, running the network in batches on a GPU rather than the CPU should counteract most of the lost speed. MCTS algorithms can readily be modified to perform state evaluations in batches, as done for AlphaGo [195], which would result in several orders of magnitude speed improvements.

7.5 Conclusions and Future Work

In this paper we have used deep CNNs to evaluate RTS game states. We have shown that the evaluation accuracy is higher than current alternatives in μRTS. This new method performed better evaluating early game positions which led to stronger gameplay when used within state-of-the-art RTS search algorithms.

While CNNs might not perform significantly better in all cases (for instance compared to Lanchester when used in ATHN-P and AHTN-F, see Figures 7.4 and 7.5), the game playing agents based on them were stronger on average. Evaluating our CNN is several orders of magnitude slower than the other evaluation functions, but the accuracy gain far outweighs the speed disadvantage.

With these promising results, coupled with the fact that modern CNNs have shown excellent results on large problem sets [127], we are confident that the presented methods will scale up to more complex RTS games. StarCraft maps are similar in size to the images these networks are usually applied to. Using an MCTS implementation based on game abstractions similar to μRTS, that allows for asynchronous state evaluations on multiple GPUs can aid in tackling these larger problems while meeting real time constraints. Moreover, policy networks may also be trained to return probability distributions over the possible moves which can be used as prior probabilities to focus MCTS on the most promising branches.

Unlike Go, however, even RTS games with professional leagues such as StarCraft do not make replays of competition games publicly available. Without a large number of high quality records, reinforcement learning techniques will likely need to be considered in future work.

7.6 Contributions Breakdown and Updates Since Publication

The bulk of this research was performed by Marius Stanescu. Nicolas A. Barriga contributed with μRTS expertise and helped running experiments, Andy Hess offered feedback and advice for using the Caffe framework and Michael Buro supervised the overall work.

The good results obtained in this article motivated us to continue exploring this avenue of research. Two updates were introduced in work described in Section 8.3. Firstly, we have evaluated the work presented in this chapter on 8×8 maps only. Having fully connected layers means that different input sizes cannot easily be accommodated, and we designed a fully convolutional network that can be used on maps of any size. Secondly, we used this new network to predict which action the Puppet Search algorithm introduced in Section 8.1 would choose for a given state, instead of the value of the state.

This policy network was only slightly weaker in play strength compared to the original algorithm, but orders of magnitude faster. Ultimately this encouraged us to explore Deep RL methods in future research, presented in Chapters 1 and 10.

Chapter 8

Combining Strategic Learning and Tactical Search in RTS Games

This chapter is a summary of work led by Nicolas A. Barriga, in which I, Marius Stanescu participated as second author. Michael Buro supervised the projects. I include short summaries instead of full articles, and focus more on placing this work in context with other research presented in this thesis.

8.1 Puppet Search: Enhancing Scripted Behavior by Look-Ahead Search with Applications to Real-Time Strategy Games

This section describes and contains parts from joint work with Nicolas Barriga and Michael Buro. It was previously published [11] at the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE), 2015.

This work introduced a new search framework, *Puppet Search*, that combines scripted behavior and look-ahead search. We presented a basic implementation as an example of using *Puppet Search* in RTS games, with the goal of reducing the search space and make adversarial game tree search feasible.

Puppet Search builds on hierarchical decomposition ideas and adds look-ahead search on top of expert knowledge in the form of non-deterministic scripts. These scripts can expose choice points to the search procedure, adding flexibility and the ability to adapt to unforeseen circumstances better than hand-crafted traditional scripted AI systems. Moreover, the branching factor will be easier to manage and by controlling the number of choice points in this process, the resulting AI system can be tailored to meet given search time constraints. Selecting a combination of a script and decisions for its choice points represents a move to be applied next. Such moves can be executed in

the actual game, thus letting the script play, or in an abstract representation of the game state which can be used by an adversarial tree search algorithm. *Puppet Search* returns a principal variation of scripts and choices to be executed by the agent for a given time span.

The algorithm was implemented in a complete StarCraft bot. During the search, look-ahead requires forwarding the game state for a certain number of frames during which buildings need to be constructed, resources mined, technology researched, units moved, and combat situations resolved. The Build Order Search System (BOSS) library [40] was used to simulate all economic aspects of the game, and a simple high-level simulation was built for troop movement. Units were grouped into squads that move from one region to another along the shortest route, towards the region they are ordered to defend or attack. If they encountered an enemy squad along the way, SparCraft [38] was used to resolve the battle.

Due to having an imperfect model for forwarding unit movement and combat, we decided against using MCTS which would heavily rely on it in the playout phase. Instead we opted for using a modified Alpha-Beta search, which required a a suitable evaluation function. A few approaches were tried, such as the destroy score assigned by StarCraft to each unit based on its costs and used in [233, 232] or LTD2 [126, 44], a basic measure of the average damage a unit can deal over its lifetime. The best results were obtained using the model based on Lanchester's attrition laws presented in chapter 5. This choice still presents a limitation, as it only models combat units and largely ignores economic incentives. An evaluation function was trained against each individual opponent played during the evaluation tournament, played against state-of-the-art bots from the 2014 AIIDE StarCraft competition.

In our experiments the average performance against all chosen opponents was similar or better than the best benchmark script, and moreover, further analysis indicated that *Puppet Search* is more robust, being able to defeat a wider variety of opponents. Despite all the limitations of the implementation used for the experiments, such as imperfect squad movement and combat modelling, incomplete evaluation function, and small variety of scripts, our encouraging initial results suggested that this approach is worth further consideration.

8.1.1 Contributions Breakdown and Updates Since Publication

The bulk of the work in this section was performed by the first author, Nicolas A. Barriga. Marius Stanescu implemented and trained the evaluation function used by the algorithm, provided suggestions for overall algorithm design, and helped write the published article. Michael Buro supervised the work.

Our first attempt at using state and action abstraction in conjunction with adversarial search for RTS games was presented in Chapter 6. The proposed

algorithm is general enough to encompass a full RTS game, but only combat-related experiments were conducted. Building state and action abstractions implies significant effort, and *Puppet Search* offers a more elegant way of implementing a similar concept by using already common action scripts as the first abstraction level. For this reason we continued future research building on the *Puppet Search* framework instead of extending Hierarchical Adversarial Search.

Improving the performance of the Alpha-Beta search component requires a more comprehensive evaluation function, as the one used in this work only accounted for combat units. For future work we proposed a function that evaluates the entire game state as described in [59] but ultimately we decided to learn such a function automatically using CNNs, which initiated research described in Chapter 7.

This first article on *Puppet Search* left a series of open questions about the impact of several design decisions such as forward model quality, search algorithm or choice points design. These questions, as well as others such as the effect of different map sizes, are answered by research described in the next section.

8.2 Game Tree Search Based on Non-Deterministic Action Scripts in Real-Time Strategy Games

This work builds on the *Puppet Search* algorithm described in the previous section, addressing some of the mentioned concerns and The experimentation is more extensive, and a wide range of map sizes was used to study the algorithms' performance when increasing the scenario complexity.

The earlier version of *Puppet Search* from Section 8.1 was tested in StarCraft, with encouraging results despite the poor accuracy of the simulator used as a forward model. All experiments in this paper were performed in μRTS – where there is direct access to the game engine – to avoid the external noise caused by an inaccurate forward model. The algorithm was directly compared against state-of-the art methods published recently for μRTS. To maintain a fair comparison, all algorithms use the same evaluation function based on fixed length playouts.

Two choice points designs were evaluated, both based on the 4 simple action

scripts that are part of the μRTS framework, each focusing on one of the 4 types of units available in the game. PuppetSingle uses a single choice point to select among these 4 scripts, generating a game tree with constant branching factor of 4. In addition to this choice point for selecting the unit type to build, PuppetBasic has an extra choice point for deciding whether to expand (i.e., build a second base) or not. Because this choice point is only active under certain conditions, the branching factor is 4 or 8, depending on the specific game state. PuppetSingle outperformed PuppetBasic on the smaller maps, while the opposite was true on larger maps. This exemplifies the importance of designing choice points carefully. They must be potentially useful – small maps do not require expanding –, otherwise they are just increasing the branching factor of the search tree without providing any benefit.

RTS game states tend to change gradually, due to actions taking several frames to execute. To use computation time efficiently and capitalize of this slow rate of change, the assumption was made that the game state does not change for a predefined amount of time and a deeper search can be performed than otherwise possible during a single frame. This approach was called a *standing plan*, and it used a generated solution (a series of choices for a script's choice points) to control the game playing agent for multiple frames, while the search produces the next solution. The longer term plan approach was beneficial on the larger maps, where action consequences are delayed and deeper search is important. On small maps computing a different plan every time instead led to better performance.

Search algorithms based on UCT and Alpha-Beta were used, with similar performance on small maps. On larger scenarios Alpha-Beta performed better, and we believe this was because MCTS algorithms usually work best for games with larger branching factors. This weakness was likely masked on the smaller maps because of the larger number of nodes that can be explored due to faster script execution.

Compared to state-of-the-art algorithms, our method showed a similar performance to top scripted and search based agents in small maps, while vastly outperforming them on larger ones. Even PuppetSingle can outperform the other players in most scenarios. In addition, on larger maps the ability to use a standing plan to issue actions, while taking more time to calculate a new plan, boosted the performance even further.

8.2.1 Contributions Breakdown

The majority of the work presented in this section was performed by the first author, Nicolas A. Barriga. My contributions were the evaluation function used by the algorithms, feedback regarding algorithm design choices and help with running experiments and writing the published article. Michael Buro supervised the work.

8.3 Combining Strategic Learning and Tactical Search in Real-Time Strategy Games

This section describes and contains parts from joint work with Nicolas Barriga and Michael Buro. It was previously published [13] at the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE), 2017.

To cope with large state spaces and branching factors in RTS games, *Puppet Search* described in the sections above focuses on state and action abstractions via action scripts. It worked increasingly well in larger problems, as it can make strong strategic choices. However, its rigid scripted tactical micromanagement is a weakness, and it showed in modes results on the smaller sized scenarios where good unit control is key to victory. More generally, algorithms that focus on state and action abstractions might suffer from limited tactical ability due to their necessarily coarse-grained abstractions. We investigated blending such algorithms with search-based approaches to refine and improve their tactical decisions. This paper's contributions are a network architecture capable of scaling to larger map sizes than previous approaches, a policy network for selecting high-level actions, and a method of combining the policy network with a tactical search algorithm that surpasses the performance of both individually.

We built on previous work presented in Chapter 7, which introduced a CNN-based evaluation function composed of two convolutional layers followed by two fully connected layers. It performed very well on 8×8 maps. However, as the map size increases, so does the number of weights on the fully connected layers, which eventually dominates the weight set. To tackle this problem, we designed a fully convolutional network (FCN) which consists of ten intermediate convolutional layers [198] without any fully connected layers, and has the advantage of being an architecture that can more easily fit a wide range of board sizes.

This network is used for two purposes, firstly as an evaluation function in conjunction with the original *Puppet Search* algorithm, and secondly as a policy network by training to predict the output of a 10 second run of this new *Puppet Search*. Since this policy will be much faster than *Puppet Search*, there will be plenty of 'thinking' time left for use by a tactical algorithm on the side. These network architectures are largely the same, only differing in the first and last layers. The policy network needs the current player as an input and it has four outputs – choosing one of the four scripts available – while the evaluation network has only two – which player wins from the given position. Training data was generated by running games between a set of bots using several different maps and starting positions.

Using a policy network for script selection during game play allows for bypassing the need for a forward model of the game, required by the original *Puppet Search*. This is usually too slow for online use in conjunction with any search algorithm. It will still be required for the offline supervised training

phase, where execution speed is less of an issue.

With the policy network running significantly faster (3ms versus a time budget of 100ms per frame for search-based agents) compared to the original search algorithm, the unused time was used to refine micromanagement during combat. A separate MCTS-based adversarial search algorithm is used to generate updated actions for all units that are within close distance to opposing units.

The evaluation network reached 95% accuracy in classifying samples as wins or losses. The resulting performance of *Puppet Search* using it was better than when using random playouts or even Lanchester based evaluation function as presented in Section 8.2 even though the network is three orders of magnitude slower. The policy network predicted the correct *Puppet Search* move with 73% accuracy and had a top-two accuracy of 95%. It was slightly weaker in playing strength but still managed to defeat all scripts and other state-of-the-art μRTS agents in the evaluation tournament.

The combined agent using a separate tactical search algorithm for micromanagement had much higher win-rates than either of its two independent components, and defeated all other state-of-the-art μRTS agents easily. To the best of our knowledge, this research was the first successful application of a deep convolutional network to play a full RTS game on standard game maps, as previous work has focused on sub-problems, such as combat, or on very small maps.

8.3.1 Contributions Breakdown and Updates Since Publication

Most of the work in this chapter was performed by the first author, Nicolas A. Barriga. My contributions were the design of the evaluation network and help with training of both evaluation and policy networks, as well as article writing. Michael Buro supervised the work.

Our next step after publishing this work was using reinforcement learning to eliminate the need for a forward model to label training data with Puppet Search. We implemented double DQN [94] with experience replay [150] and used the evaluation network to provide a non-sparse reward. Unfortunately, in all of our experiments the network converged to always selecting the script with the best average performance, regardless of the game state.

We believe that the main culprit is the reward mechanism. The virtual rewards generated by the value network are small and potentially noisy at the start of the game, when good action choices have the largest impact. Towards the end of the game the rewards are more accurate, but action choices are largely irrelevant. Any action (script choice) is able to successfully finish a game that is very nearly won, and conversely, if the position is very disadvantageous, no script choice will be able to reverse it. An attempt was made to fix this issue by starting the learning process with states closer to the endgame

and progressively introducing states from earlier in the game, but without success.

The drawback of requiring a forward model is also present in most tactical search algorithms, for example the MCTS variant we used for blending in this research. Using machine learning for tactical decisions would eliminate this requirement, but had limited success even on simple scenarios previously [235, 221]. Recent research based on integrating concepts such as communication [209], bidirectional recurrent neural networks [171] or value decomposition networks [211] promises stronger tactical networks for strategy games and draws even more attention towards using reinforcement learning.

Consequently, for future research I continued to explore the reinforcement learning paradigm for use in RTS games and techniques that help in complex environments with sparse rewards: reward shaping [52], curriculum learning [19] and hierarchical reinforcement learning that use auxiliary rewards for the lower levels [101, 128].

Chapter 9

Case Study: Using Deep RL in Total War: Warhammer

The following work was done as part of a 7-month internship at Creative Assembly, the UK games studio behind the popular Total War franchise. The aim of this project was to apply some of the latests insights from deep RL research to RTS games. The 4-1 victory of AlphaGo 2 months previous to starting this project provided additional support and incentive for bringing the games industry and AI research closer.

After successfully using deep CNNs for state evaluation in the basic, research oriented μRTS framework, and experimenting with DQN-based methods for small combat scenarios in the StarCraft: Broodwar game, I was confident that deep RL can be used in even more complex environments. This internship presented an opportunity to tackle more difficult multi-agent problems and to get acquainted with the games industry's perspective on such challenges. As well as a stepping stone to further research in the MARL domain, using an off-the-shelf game engine for this project offered a new outlook on blending academic research with the game production process.

9.1 Introduction

Inspiration for this project stemmed from the recent success of AlphaGo [195] which won 4-1 against one of the strongest human Go players in 2016 – illustrating the effectiveness of combining MCTS and deep learning techniques. For AlphaGo, CNNs were trained to shrink the prohibitively large search space of the game of Go in two ways: first, a *policy network* was trained, using both supervised and RL techniques, to return a probability distribution over all possible moves. These results were used to focus the search on the most promising branches. Second, MCTS state evaluation accuracy was improved by combining playout results with evaluations produced by a *value network*.

The work presented in the previous chapter focused on this second part – state evaluation. The results were very promising, and we continued by ex-

tending this work and investigating learning policies for unit control in small scale StarCraft: Broodwar scenarios. Good progress was made, but even better results using similar methods were published first in [235] and then then in [171], and consequently we were not able to publish our findings. Instead, we took advantage of the accumulated experience and aimed to apply it for battle AI in the popular Total War game franchise. Battles in Total War can pose much more difficult challenges, since the complex movement and targeting that is crucial to games such as μRTS and StarCraft is affected by additional mechanics: units can get tired, a defender's orientation with respect to its attacker is crucial, there are effects that reward charging an enemy but discourage prolonged melee (cavalry) or the other way around (pike infantry), and morale is often more essential for obtaining victory than troop health. As a consequence, strategies such as pinning, refusing a flank while trying to overpower the other flank and obtain rear attacks on the enemy are required. Cooperation between units is much more important than in games such as StarCraft, and the built-in game AI uses complex behaviors and a portfolio that contains multiple strategies. As a result, besides good reactions, human or AI players need to employ complex behaviors as well to consistently defeat this built-in game AI.

Automatically training agents to achieve diverse behaviors is a natural fit for the Total War battle AI. In Total War games, every time the players choose to fight a battle during the turn-based campaign, they usually have three options: 1) flee to avoid the battle, 2) let the built-in AI autoresolve the conflict for them and 3) or pause the game and proceed to a arena-like environment where they can fight the battle in real-time. A screenshot of how such a battle might look can be seen in Figure 9.1. Considering the duration of and the resources available for the internship, we chose to address this multi-agent battle aspect of the game rather than the challenges presented by the campaign part for the following reasons:

- As the number of possible agent actions increase, the learning process becomes much more difficult and more data is needed for sufficient exploration. In battles, agents could have a few move actions (e.g., four or eight cardinal directions) and several attack actions either generated by some scripts (e.g., attack closest, weakest etc) or by direction (e.g., attack North). However, for a campaign game, just considering the different structures one can build in one's settlements might spawn tens of action by itself.

- The earlier a training episode ends and an agent can receive its reward, the easier it is to explore and learn good behaviors. Outcomes of campaign games are much more delayed than those of single battles. For instance, the impact of diplomacy decisions (such as declaring war on a neighbor) would be particularly hard to quantify. Just using learned policies directly to make battle decisions at run-time is likely to work

Figure 9.1: Example battle scenario in TotalWar: Warhammer.

well, while for campaign games we would need to use look-ahead search to explore the consequences of our decisions and offset the longer delay in observing their impact.

- The input size is larger for campaign games. Moreover, training by playing a hundred thousand single battles is more feasible than acquiring the same number of complete campaign games.

While achieving strong campaign AI might be a more difficult task, battle AI offers a diverse set of challenges that require spatial reasoning and cooperation between the agents, as previously mentioned. Some of the most commonly encountered are: deciding when to pull out from a melee and when to switch targets, positioning of the range units, avoid crossing paths between ally units, maintaining a coordinated unit front when approaching the enemy, or pinning enemies while creating superiority elsewhere.

9.2 Implementation

The goal of this project was to learn control policies for agents in cooperative-competitive environments. RL, in particular, represents a natural fit to learn adaptive, autonomous and self-improving behavior in a multi-agent setting. CNNs have made it possible to extract high-level features from raw data, which enabled RL algorithms such as Q-learning to master difficult control policies without the help of hand-crafted features and with no tuning of the architecture or hyper-parameters for specific games [149].

9.2.1 Network Architecture

The network architecture chosen is similar to the AtariNet model, and used three convolutional layers followed by a fully connected one and output layers. Lower dimensional input was used instead of raw pixels, as coarser grid representations worked well in previous experiments on StarCraft combat scenarios. The network weights were learned via Q-learning, while agents played games against the built-in AI.

Due to the limited time and resources available the Independent Q-Learning (IQL) [223] paradigm was chosen. In IQL, the current agent is considered to be learning while all other agents are treated as part of the environment. Thus, the multi-agent learning task is decomposed into simultaneous single-agent problems.

For each unit, the input state consisted of a number of 128×128 grid-like, local feature planes centered around the unit:

- 6 of them were Boolean values indicating own and enemy presence on each grid square, and the unit types grouped in 4 categories: spearmen, other melee infantry, archers and cavalry. There is a rock-paper-scissors interaction, and similarly priced units counter and are countered by other unit types. Usually swords or axes beat spears in melee, spears are the best counter to cavalry and cavalry units can cause great problems to melee troops. Archers can deal damage from a distance to most units except perhaps the slower, heavier armored ones, but usually suffer in melee.

- 4 more feature planes contained values indicating hit points, morale, orientation and stamina. Only one unit type was used for each of the 4 categories, so there was no obvious need of adding more features such as damage, armor or defence values as they do not change during the course of the battle and can be absorbed into the unit type.

- There were as many Boolean layers as enemy units, used for masking them and correlation with the attack enemy order.

- There were two more feature layers containing minimap information for both armies. For these layers, the original map unit information was scaled down to 128×128, and normalised such that each cell value was equal to the number of units within divided by total number of alive units. A value of 1 was added to the cell containing the acting agent, in the allied minimap. These minimap layers were included to provide the agents information about the global state. All other features were normalised by their maximum possible values.

The possible actions for each unit were chosen as: no-op, move in one of the 4 cardinal directions, and an attack action for each enemy unit in the

scenario. The attack actions corresponding to units that are not shown in the local feature planes were masked out.

9.2.2 Reward Shaping

Because team reward signals do not directly relate to individual agent's actions, individual reward functions are commonly used for IQL and were used here as well. Reward shaping via potential-like functions was chosen, as it is one of the few methods that guarantee to preserve optimality w.r.t. the initial objective [52, 57]. The score function ϕ is a weighted sum of the unit's current health, morale and stamina values as well as the damage inflicted upon enemy units to both health and morale from the start of the game and a bonus term for each routed or killed enemy unit. The reward signal at time step t, for unit i was $R_t^i = \phi(S_t^i) - \phi(S_{t-1}^i) - r_{step} + r_{outcome}$ where r_{step} is a small positive term to discourage time wasting and $r_{outcome}$ is 1 if the battle ended with a victory and 0 otherwise.

Learning was evaluated objectively by win ratio against the built-in AI on the highest difficulty, rather than by subjective human players. As a note, the games industry is often more concerned about players having a fun experience and prefers a human-like, believable AI to stronger but potentially less polished behavior. Avoiding behaviors such as switching orders frequently, units crossing each others' paths, erratic movement patterns at the start of the game or not maintaining a coordinated unit front can often be more important than optimality and winning at all costs. Adding extra terms to deal with these concerns via reward shaping is not difficult, and it was a major argument for choosing this form of providing rewards.

9.2.3 Experimental Setting

Although troops could be concealed in forests, for example, fog of war does not play as big a role in Total War battles compared to other RTS games. Thus, dealing with partial observability was out of scope of this project, and there was no unit vision limitation or fog of war mechanism. The proposed method could be extended to work with imperfect information, and several ideas for doing so are mentioned in section 9.5. For simplicity, small flat maps were used, without terrain features such as hills, forests or river crossings.

Battles in Total War are driven by units: groups of specific troop types that can be deployed in formation. In our experiments, up to 6 units were used per army, each unit containing 40 to 100 soldiers depending on its type – infantry units usually contain more soldiers than cavalry ones, for example. In Total War battles orders are given on a per-unit basis, and the game engine deals with the movement and attack animations of the individual soldiers within the unit, to maintain coherence.

Actions were emitted every 3 seconds for all alive units, which was also the

default setting for the built-in AI. For these settings battles were often concluded in around 5 minutes or real-time play. To avoid dithering, we forcefully ended games after 7.5 minutes of real-time play, which translates to up to 150 orders per unit.

The game ran on Windows machines, and the ZMQ distributed message queue framework was used for communication with the machine learning server running on Linux and using PyTorch. Due to the large number of resources needed for running the game engine, only up to 8 game instances could run simultaneously on a single machine, and only if graphics were completely disabled. Usually, on the scenario sizes described above, up to 9k - 10k games could be played per machine (Intel Core i7 CPU and Nvidia GTX 1070 GPU) in 24h.

9.2.4 Models and Training

A network architecture similar to the ones used in Atari research and early experiments in the StarCraft II Learning Environment (SC2LE) [236] was used. The 128×128 feature planes were first processed by three convolutional layers. The extracted map representation was then passed through a fully connected layer with 256 units. The Q values were represented using value and advantage functions as recommended by the dueling architecture [237].

To reduce the number of learnable parameters network weights were shared between all agents. Training battles were fought against the built-in AI on the highest difficulty setting. The learning algorithm was based on DQN [150] with the dueling architecture update and multi-step returns. The replay buffer was chosen to contain only the most recent 1000 games worth of experiences, to help with stability issues. Close to default settings were chosen for the other DQN parameters, with brief tuning on a scenario where each player controls 2 melee units.

9.3 Results

Results were very good for scenarios where the DQN player controlled 1 or 2 units, the DQN agents defeating the built-in AI in over 90% of the games. Complex behaviors often used by human players were learned, such as "cycle charging" which works as follows. Cavalry units have a large charge bonus that helps deal significant damage upon impact, but wears down after a few seconds. Since they are often overmatched in the ensuing melee, a good strategy is to retreat, regroup and repeat charging for as long as possible. When controlling a weaker melee unit and a cavalry unit against a stronger enemy melee unit, the system learned to use the melee unit to pin down the enemy and only then use the cavalry troop to manoeuvre and charge with the cavalry from the enemy's flank or rear. Doing so heavily deteriorates enemy morale and often

Total War battles are won or lost by reducing morale and causing the enemy to flee, rather than by killing all enemy soldiers.

An example of this behavior can be seen in the following replay [1]. In this and all following videos shown in this chapter, the agent controls the highlighted troops with green health bars, while the enemy has red health bars. Here the enemy commands an elite axe melee unit. The agent controlling a weaker spear melee unit pins down the enemy and the cavalry unit charges it repeatedly in the rear.

When using more than 3 units per side, learning with DQN was very unstable, probably due to the non-stationary aspect of the environment. The learning and exploration of other agents adds noise and instability, and the environment is changing so fast that games in the experience replay quickly lose relevance. Hence, the learning algorithm was switched to A3C, an on-policy actor-critic framework. As a consequence learning was more stable, even though less efficient w.r.t the number of games played due to the lack of experience replay.

Using a set of network weights for each unit type (spear, melee, cavalry, ranged) worked better than sharing the same set of parameters for all units. The algorithm plateaued at over 82% win rate on a symmetric 3v3 scenario with mixed units, after 70k games played. Over 73% win rate was achieved when scaling up and training from scratch on 6v6 scenarios, but learning required more than twice the time, at around 150k games. Further increases of army sizes seemed achievable, but were not attempted due to the limited time and computational resources available.

The following examples show battles with agents trained via A3C, the next two using agents that learned on 3v3 scenarios with mixed units. In the first video [2] each side controls 3 identical melee units and the agents display a coordinated unit front while closing the distance to fight. They use flank attacks and better positioning to bring more troops in contact, fighting on the frontline. In the second video [3] there are 2 melee units and 1 ranged unit per side. The ranged agent avoids the incoming melee units and takes out the enemy ranged unit, while the melee units surround the enemy melee and finish them off with subsequent help from the ranged agent. Finally the last example [4] shows agents trained on the 6v6 scenarios controlling 4 melee and 2 ranged units. The melee units out-manoeuver the enemy and obtain a better front, while the ranged units help focus damage on the same melee units. The leftover enemy ranged are easy to deal with afterwards.

Compared to training directly on the last scenario, learning was faster when using a curriculum based approach. Army size was extended progressively from 1 to 6, and for each scenario the hit points of enemy units were gradually

[1] https://drive.google.com/file/d/1ZnJWhU3FiUYZ7ZIblmp_jT-FPItuuJGP/view
[2] https://drive.google.com/file/d/1SiQH5FNEEVWyl1T3wUoK8YlwTTL-5tXt/view
[3] https://drive.google.com/file/d/1dd8On9o3d_gxkWjYbRPgs4_QrnBZm1ow/view
[4] https://drive.google.com/file/d/1etkbNw0XEqz1XScTtQmFkSFDvey68jA-/view

increased from 60% to 100%. Scenario switching was done when the algorithm reached 70% win rate on the current task, up until reaching the final task. With these settings a higher win rate of 77% was achieved for the 6v6 scenarios, using fewer games for training – 110k.

9.4 Hierarchical Reinforcement Learning

The results described above confirmed that the proposed method, even with basic architecture choices and without extensive hyperparameter search, worked well for the intended purposes. The last month of the internship was then aimed at investigating ways of obtaining similar results using less samples/games or at scaling towards scenarios with more agents.

Hierarchical reinforcement learning (HRL) was chosen as a potential answer to both concerns, as it is a promising approach to expand traditional RL methods to work with more difficult and complex tasks. In HRL several levels of policies are trained to control agents at increasingly higher levels of spatial, temporal and behavioral abstraction. In this hierarchy of policies the lowest level applies actions to the environment, and the higher levels learn to plan over a larger region or extended time period.

9.4.1 Implementation

We decided against using an option-critic architecture [6] as these approaches can have difficulties such as learning sub-policies that terminate every step or learning policies that run for the length of the whole episode. A better fit for industry use would be methods that use auxiliary rewards for the lower levels, similar to [101, 123, 128, 226]. Using hand-crafted rewards based on prior domain knowledge would be easier to implement and used to control the resulting agent behaviors.

A two-layer structure was used, with the higher level policy making decisions on a coarser layer of abstraction and choosing goals to be implemented by the lower level policy, in a similar fashion to [66]. The higher level policy observed the current state every $k = 10$ steps and chose a goal to be achieved from *approach, outflank, attack, disengage* and an enemy unit w.r.t which this particular order should be considered. For example *approach unit x* is be successful when the distance between the current unit and unit x is less than a particular threshold; disengage aims for the opposite, outflank relies mostly on the relative orientations of the two units and attack is assessed based on inflicted damage to target unit.

The lower level policy observed the state and the goal every time step, and produced an in-game action – from the same action set as in previous experiments – which was applied directly to the environment. Training this lower level was done via intrinsic rewards provided by the higher-level controller. For this experiment the rewards were again designed using potential based

shaping. For example, the *approach unit x* goal for unit u used a potential of the form $\phi(S_t^u) = w_1 * HP_u + w_2 * morale_u - w_3 * (d(u,x) - 10)^2$, with the first two terms discouraging recklessness and the last term based on the distance between the two units. The default rewards provided by the environment are summed up over the 10 steps and used to train the higher level policies.

9.4.2 Results

Training both high and low level policies jointly did not yield good results for the 6v6 scenario even after a few days of learning (roughly $200k$ games), the win rate never going over 40%. To investigate, we pre-trained the low level policy first and then fixed the corresponding network parameters and trained only the high level policy. The low level policy learned how to accomplish the disengage and approach objectives in less than 5000 games, the outflank objective in 8000 games and the attack objective in 25000 games. The achievement of the first three objectives was judged based on the final relative positions of the involved unit, while the attack was considered successful if the total morale and hit point damage inflicted exceeded the received amount by a given margin.

Even with a pre-trained and fixed lower level, the high level was not able to surpass 50% win rate even after playing a comparable number of games as the non hierarchical version. This suggests the designed goals and their hand-crafted auxiliary rewards did not map very well to the main task to be solved – winning the battle. The number of possible goal actions $N_g = 4 \times N_{alive\_enemies}$ is higher than the number of low level environment actions $N_a = 4\,moves + N_{alive\_enemies}$ except when there is only one enemy unit left. The difference is largest before engaging and killing enemy units, and that is when smart maneuvering is most important! Exploring via ε-greedy action selection might require a higher number of games and, moreover, game result is very sensitive to one bad high-level decision. One poorly timed withdraw order can expose one's flank to being rear charged and that may lead to chain routing of the whole army and a subsequent defeat.

9.5 Conclusions and Future Work

While basic, the proposed approach worked well for less complex scenarios. Scaling up to the maximum number of unit in a Total War battle (20 per side) might prove challenging, however. IQL suffers from instability caused by non-stationarity introduced by learning and exploration of other agents and increasing the number of units will accentuate this effect.

In addition, more computation time would be required, and more games would need to be played. Hierarchical RL and other methods that are more data efficient should help, especially in conjunction with environments where obtain samples is not cheap. One idea is to use *world models*, which learn compressed spatial and temporal representations of the environment and can

be trained relatively quickly in an unsupervised manner [87]. Then compact agents can be trained, even entirely inside their own hallucinated dreams generated by the world model. The authors show that such policies transfer surprisingly well to the actual, real environment. Learning how to decompose the team reward should help as well, and there are methods that proved successful in partially-observable cooperative multi-agent scenarios with a single joint reward signal [211].

In a multi-agent hierarchical framework, high level decisions that are followed for a fixed number of steps can have multi-modal returns. Depending on what its allies and enemies do, following an order that has a large expected value might often lead to negative rewards. In other words, there might be many risky choices and currently the games industry is not in a position to explore high risk AI designs. Distributional RL [17] is a paradigm that has been proved to help preserve multimodality in value distributions and lead to more stable learning.

Finally, a few assumptions were made that could be addressed by a larger project, most notably about the imperfect information aspect of the game. To deal with these, there are a few extensions that have been considered but were out of the scope of this project's timeline:

- Fictitious self-play can be combined with deep reinforcement learning to learn stronger strategies in *imperfect information* scenarios. This has been shown to converge reliably to approximate Nash equilibria in cooperative games [154] and poker [103] while simple greedy or average deep Q-learning strategies did not. Stacking multiple input frames along the depth dimension, using gated architectures such as LSTM and GRU, could also help in this respect.

- *Policy distillation* can be used to extract the policy of a reinforcement learning agent and train a new network that performs similarly while being dramatically smaller and more efficient [179]. Results show agents with 15 times fewer NN parameters performing on par with the original networks. Furthermore, the same method can be used to consolidate multiple task-specific policies into a single policy. The implied run-time speedup and memory reduction is especially important for resource intensive games that might have to run on budget hardware.

- The model could be extended to learn the *behavior of opponents* in addition to a good policy [98]. A second, separate NN is used to encode observations of opponent actions, and different strategy patterns are discovered by this enhanced model without extra supervision.

9.6 Contributions Breakdown and Updates Since Publication

The work presented in this chapter was performed by Marius Stanescu during an internship at Creative Assembly, UK. Guidance in respect to the game engine, existing codebase, and overall project management via weekly planning meetings was provided by Andre Arsenault, Scott Pitkethly and Tim Gosling, from Creative Assembly. Advice with machine learning design was offered by Michael Buro. Besides implementing and experimenting with the DQN, A3C and HRL agents described, other tasks completed during this project include:

- Writing a basic module that acts as a bridge between the C++ game engine and the machine learning framework – PyTorch. A structured, grid-like representation of the game state is provided to the machine learning side, and unit commands are received back and then executed in the game. Substantial effort was required to create and control custom scenarios, to disable nonessential game features and speed up game running, as well as to reduce its resource footprint and enabling several instances to run on the same machine.

- Implementing a server-client framework that connects the machine learning code (server) to multiple instances of the game running on one or multiple machines (clients). As the game was not designed to run thousands of games in this continuous regime, there were a few memory leaks and exceptions that required handling.

- Helping a Creative Assembly employee integrate the above framework in an existing overnight testing tool that used idle machines to run more experiments.

We found that designing good individual reward functions is very important to the overall learning stability and might require extensive tuning to obtain impressive results. Using team reward signals and then learning how to assign the credit might prove a more elegant and simple option. During this internship another paper was published that dealt with reward decomposition in multi-agent environments [174]. Together with the already existing VDN [211], this new QMIX architecture showed the advantages of such decompositions in mixed cooperative-competitive scenarios and convinced us to add similar concepts to our next research.

Off-policy algorithms, often based on some form of Q-function learning, generally exhibit substantially better sample efficiency than on-policy actor-critic or policy gradient variants [83]. This was the case in our project as well, and was later supported by research such as [156, 155]. On-policy training has generally been attractive in the past and at the time of this project because is it usually more stable. However, we think that better results can be obtained

by taking advantage of subsequent progress in robust, general off-policy RL methods [156, 15, 67].

Consequently, our next efforts use off-policy algorithms and also learn how to decompose the team reward among agents. Even though time constraints prevented reaching good results with HRL in this project, we believe that spatial and/or temporal abstractions are needed for more complex or difficult environments. The following chapter describes research that uses these insights and tries to overcome inefficiencies of standard multi-agent Q-learning methods by exploiting existing spatial action correlations.

Chapter 10

Multi-Agent Action Network Learning Applied to RTS Game Combat

This chapter describes work done by Marius Stanescu and supervised by Michael Buro. It is accepted for publishing at the Workshop on Artificial Intelligence for Strategy Games, part of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE), 2018.

Abstract

Learning good policies for multi-agent systems is a complex task. Existing methods are often limited to a small number of agents, as learning becomes intractable when the agent number increases considerably. In this paper we describe Spatial Action Decomposition Learning that tries to overcome inefficiencies of standard multi-agent Q-learning methods by exploiting existing spatial action correlations. We apply our method to real-time strategy (RTS) game combat scenarios and show that Spatial Action Decomposition Learning based systems can outperform handcrafted scripts and policies optimized by independent Q-learning.

10.1 Introduction

As multi-agent decision problems are ubiquitous, building better AI agents that work together has numerous real-world applications, such as city transport optimization, stock trading, advertisement bidding, multi-player online gaming, and coordinating robot swarms. Multi-agent reinforcement learning (MARL) [169, 31] is a popular solution paradigm in this research area that considers a set of autonomous agents that share a common environment and jointly optimizes a single team's reward signal accumulated over time. While

applying standard RL techniques like Q-learning to multi-agent settings seems straight-forward, difficulties arise from the combinatorial explosion of the joint action sets and non-trivial interactions with the environment and other agents – which can be cooperative or adversarial or mixed (e.g., opposing teams). In addition, partial observability and communication constraints require decentralised policies that only depend on local agent observations. This also helps dealing with the exponential growth of the joint action space when increasing the number of agents. We can leverage extra information that is usually available in simulations or lab setting by learning these decentralised policies in a centralised fashion using extra state information that is hidden from agents at runtime. This approach has become very popular lately [115, 62, 174].

A challenging aspect of decentralization is to find an effective representation of the action-value function in which we need to integrate the effects of all agents' actions in a centralised action-value function Q_{tot} which depends on the global state and global joint actions. But such functions are difficult to learn in the presence of many agents and extracting the decentralised individual agent policy (one agent chooses one action based on individual/partial observation) is not straightforward.

In the next section we discuss in more detail related work that tackles multi-agent Q-learning in various ways – ranging from centralized learning, over simple Q-function decompositions, to learning expressive networks for mixing individual Q-functions. We then describe our novel learning approach which is based on the observation that agent actions are often spatially correlated (e.g., nearby agents frequently execute similar actions), such as moving in the same direction, or collaborating to achieve a local goal such as defending a choke point. We then present experimental results for our spatial action decomposition method – applying it to the popular multi-agent learning domain of 2-team combat in real-time strategy (RTS) games with up to 80 vs. 80 units. Finally, we conclude the paper with summarizing our findings and discussing future work.

10.2 Background and Related Work

10.2.1 Independent Q-Learning

Arguably the easiest and most commonly used learning method for multiple agents is Independent Q-Learning (IQL) [223]. It considers all other agents as part of the environment and decomposes the multi-agent learning task into simultaneous single-agent problems. IQL suffers from instability caused by non-stationarity introduced by learning and exploration of other agents [133], and consequently loses the convergence guarantees of Q-learning. As a concrete example, [46] show that independent Q-learners cannot distinguish team mates' exploration from stochasticity in the environment, and fail to solve even an apparently trivial, 2-agent, stateless, (3×3)-action problem.

While there are ways of improving IQL's stability [63], to work well it requires individual reward functions because uniform team reward signals do not directly relate to individual agents' actions. Reward shaping is difficult and few methods guarantee to preserve optimality w.r.t. the initial objective [52, 57]. A preferable, more general approach is to learn how to decompose the team reward. Still, in practice, IQL is an unexpectedly strong benchmark method, even in mixed cooperative-competitive MARL problems [136].

10.2.2 Centralised Learning

Alternatively, the joint action Q-function Q_{tot} can be learned directly. This avoids the non-stationarity problem and can lead to better coordination and results, at the cost of poor scaling performance. Some methods are partially centralised, using one or more centralised critics to guide the optimisation of decentralised policies in an actor-critic paradigm [62, 86, 136]. To work well, these methods require additional information to be exchanged between agents (e.g., CommNet [209] or BicNet [171]). Furthermore, such on-policy methods are usually sample inefficient.

Using a team reward signal makes credit assignment challenging, even for simple problems. For example, in a 2-player soccer game with the number of scored goals being the team reward, the agent who is worse at scoring sometimes ends up failing to learn to shoot at all, as its exploration would impede its teammate [97].

Coordination graphs have been used to decompose the global reward into a sum of local agent rewards [84], but the method requires solving a linear program and message passing between agents at runtime. COMA [62] is an actor-critic method that uses a counterfactual baseline to marginalise out a single agent's action, while keeping the other agents' actions fixed. Another idea is to transform multiple agent interactions into interactions between two entities: a single agent and a distribution of the other agents [252].

10.2.3 Value Decomposition

A more elegant way of solving the credit assignment problem is to use a value decomposition network (VDN) to represent $Q_{tot}(\tau, u)$ – where τ and u are the joint action-observation history and the joint action – as sum of individual Q-functions Q_i which depend only on agent-local observation histories [211]:

$$Q_{tot}(\tau, u) = \sum_{i=1}^{n} Q_i(\tau^i, u^i; \theta^i).$$

The network learns how to assign the team reward signal to Q_is implicitly, without shaping or global state information.

One disadvantage is that the VDN representation of Q_{tot} is limited by the addition, because agents' interactions are usually more complex. QMIX

addresses this issue by replacing the sum operation with a mixing network that combines all individual Q_i into Q_{tot} in a complex, monotone, and non-linear fashion informed by the global state information during training [174]. Access to the global state is not required after training because due to monotonicity, the arg-max performed on Q_{tot} yields the same result as individual arg-max operations on each Q_i.

While more natural, these methods still suffer when handling larger numbers of agents as Q-learning becomes infeasible due to noise accumulation caused by many exploratory actions [47].

10.2.4 Abstractions and Hierarchies

To address the scaling issue and improve sample efficiency the value decomposition networks can be combined with hierarchical decomposition. This can be done by considering temporal abstraction layers [137] that speed up learning by dividing the work of learning behaviors among multiple policies and exploring the environment at higher levels more efficiently. Alternatively, in suitable domains spatial abstractions can be used to speed up learning. In feudal reinforcement learning [50] the state space is hierarchically subdivided into increasingly smaller regions at each level of abstraction – similar to quad-trees in the grid world pathfinding example example they discuss. Each level has an associated Q-function that is trained by giving lower levels credit for achieving higher level goals. The experimental results indicate that feudal reinforcement learning outperforms classic non-hierarchical Q-learning in their domain. The method we present in the next section is also based on spatial decomposition. However, its objective is to generate concurrent actions for multiple agents and to effectively learn agent policies despite huge combinatorial action spaces.

10.3 Spatial Action Decomposition Learning

In this work we focus on the MARL challenge of increasing the number of agents and the resulting combinatorial explosion of the joint action sets and interactions with other agents present in the environment, friend or foe. As previously mentioned, IQL is a quick and simple solution, but does not guarantee convergence and is poorly suited for modelling all interactions between agents. On the other hand, fully combinatorial approaches are better at handling coordination but do not scale well with the number of agents.

One approach that showed good results is to estimate the join action-value of a team of agents as a linear [211] or non-linear [174] combination of individual, per-agent action-values. These per-agent values condition on agent-local observations, which is required in a partial observable environment. [174] use hypernetworks to integrate the full, global state information into the joint action-value during training.

10.3.1 Spatial Action Decomposition Using Sectors

We build on the value decomposition idea, but as we focus on the number of agents as the main challenge, we choose to work in a fully observable environment. As such we do not have to condition on agent-local observations. We decompose Q_{tot} into individual values similarly to VDN and QMIX, but use the full, global state both during training and evaluation. To handle the large number of agents and the difficulty introduced by their exploratory actions, we use a spatial decomposition. We add a *sector* abstraction, making the assumption that the joint action value can be decomposed into value functions across sectors – disjoint areas of the map – instead of individual agents. We use a simple sum and leave more complex decompositions such as QMIX to future work, as experiments in [174] show that non-linear value function factorisation is often not required for scenarios with homogeneous agent types.

After the sector actions are chosen by the network, each sector is responsible for emitting low-level actions for each agent present within, through a separate mechanism. This is a much simpler problem, and methods that do not scale as well (e.g., search algorithms if there is a forward model) or that are not very complex but very resource inexpensive (e.g., action scripts) can be used.

Each sector's Q-function Q_{S_i} will be learned implicitly by the network and will not benefit from any reward specifically given to sector S_i or to any individual agent. Strictly speaking Q_{S_i} is more an utility function than a value function because it does not estimate an expected return by itself. For simplicity, however, we will continue to call both Q_{tot} and Q_{S_i} value functions.

Each sector chooses its action greedily with respect to its own Q_{S_i}, and the joint action Q_{tot} is chosen by maximizing $\sum_i Q_{S_i}$. The maximization of Q_{tot} can now be performed in time linear in the number of sectors, which will usually be much smaller than the number of agents. For this method to work well, the sectors dimension should be large enough such that at least a few agents are present in most sectors.

10.3.2 Network Architecture

To reduce the number of learnable parameters it is common to use a neural network for each agent and to share its weights between the agents. This approach can be taken with sectors as well, but since we have access to the global state it is more natural to use convolutions to keep the number of parameters low.

A deep ConvNet computes higher and higher level features layer by layer, with the same or possibly different spatial resolutions. It's a very good tool for a sector-based approach: it starts with higher resolution maps with low-level features and produces lower and lower resolution maps but with increasingly relevant features until reaching the target sector granularity.

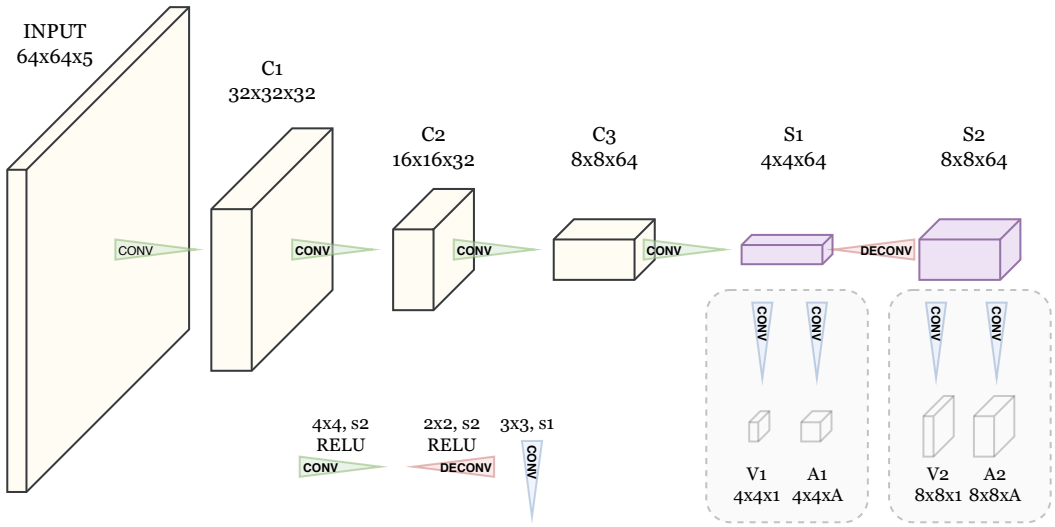


Figure 10.1: Layout of the network architecture. More deconvolutional layers can be added to increase the sector granularity.

An example network used for a grid-like 2-player combat scenario is shown in Figure 10.1. There are 5 input feature planes for a 64×64 grid map. Three of them are Boolean values indicating terrain and own and enemy unit presence on each square, and two planes represent the hit-points of the two armies. More details about the environment used for experiments can be found in the next section.

Four convolutional layers with 4×4 stride-2 filters, each followed by a ReLU nonlinearity and using batch normalisation, are used to extract features and gradually halve the map resolution, obtaining activations $C_{1-3}$and S_1. For simplicity, at a given resolution level we considered only square sector configurations of the type $W \times W$, resulting in W^2 total sectors. Using a granularity of 2×2 sectors – each 32×32 cells – was not enough to execute complex maneuvers. Consequently, we start with the 4×4 sector representation of S_1, each sector being responsible for agents within 16×16 cells on the original map.

In certain situations it might be beneficial to have even more precise control. For this purpose we can apply the standard process of deconvolution, also known as transposed or fractionally-strided convolution [54], to generate features for more, smaller sectors. For example, in Figure 10.1 S_1 is upscaled by a factor of 2 to the 8×8 representation of S_2 using 2×2 stride-2 filters.

Either S_1 or S_2 can be used to obtain Q_{S_i} values, depending on the desired sector granularity. The Q functions are represented using value and advantage functions $Q(s, a) = V(s) + A(s, a)$ as recommended by the dueling architecture [237]. More robust estimates of the state values can be obtained by decoupling them from the necessity of being attached to specific actions. For S_1 the value function denotes simply how good the current state is for

each of the $4 \times 4 = 16$ sectors, and a corresponding $4 \times 4 \times 1$ block V_1 can be obtained from S_1 by simply using a convolutional layer with 1 filter of 3×3, stride-1. The advantage function tells how much better taking a certain action for any given sector would be compared to the other actions. Using a number of filters equal to the number of possible actions A for a sector results in the corresponding $4 \times 4 \times A$ block A_1. The argmax over the last dimension gives the action with the highest advantage value for each of the 16 sectors.

The same output head filters can be applied to any resolution S_i layer to obtain Q-values for a desired sector granularity. Actions can be emitted based on the values from any of these layers, either by choosing a specific resolution beforehand, by taking the max across all resolutions, or even by adding a separate network component responsible for learning which sector granularity to pick given the current state.

Increasing the number of sectors provides more accurate control over the agents, but also increases the difficulty of the credit assignment and slows the learning process. Using fewer sectors and the resulting ability to specify large scale actions should make the exploration for actions leading to large rewards more efficient, and unnecessary micro-management could be avoided for map areas that are largely empty or that do not require complex behaviors.

In this work we focus on spatial decomposition, and opt for the simple fully convolutional design described above. Although more sophisticated network blocks could be designed, and even though temporal abstractions are compatible with our method, they are not the focus of this paper. Stacking multiple input frames along the depth dimension, using gated architectures such as LSTM and GRU, or learning agent policies that implement the provided sector actions are left for future work.

10.4 Experimental Setting

In this section we describe the combat scenario problem to which we apply Spatial Action Decomposition Learning. We provide details of the state features and training methodology and evaluate our method's performance in comparison to independent Q-learning.

10.4.1 Environment

Real-time strategy (RTS) games are now a well established benchmark for the RL community. They offer more difficult multi-agent challenges compared to previous environments such as Atari games. In particular, RTS game combat scenarios are a popular evaluation method for MARL algorithms, offering mixed cooperative and competitive multi-agent environments. For instance, the StarCraft II Learning Environment (SC2LE) [236] is a popular platform for RL experiments. However, for our experiments we chose the simpler but faster many-agent (MAgent) environment [256]. It supports up to a million

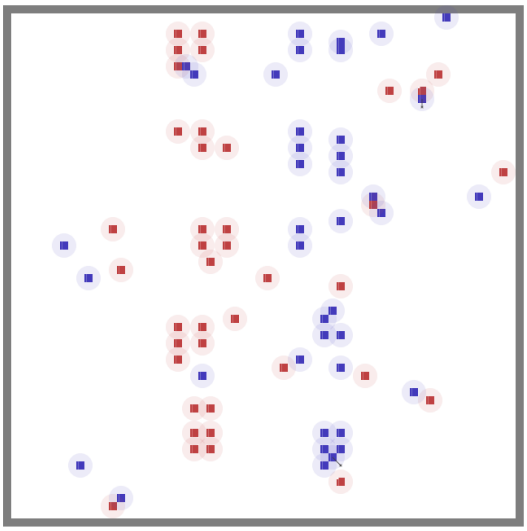


Figure 10.2: Randomly generated MAgent scenario, 64×64 map with 40 units on each side, loosely split into groups.

agents on the same map while being much less computationally intensive compared to SC2LE. Moreover, it provides environment/agent configurations as well as a reward description language that enables flexible environment design.

In this work we focus on a centralised micro-management scenario in which two groups of identical units fight on a grid-world map. The units of the first army are controlled by the learning method. The opposing units are controlled by a scripted algorithm that moves towards and focuses fire on its closest enemies. The initial placement of the units for each army is randomized at the start of each game: around two thirds are split randomly among a random number of 2 to 5 orderly groups, while the remaining units are randomly scattered on the map. An example scenario is shown in Figure 10.2. This setup is designed to mimic combat scenarios commonly occurring in popular RTS games such as StarCraft II. The small attack range compared to the size of the map and the number of units and clusters promote smart group maneuvering which is notoriously hard for agent-centric learning.

We adopt MAgent's default combat settings: at each time step agents can either move to or attack any cell within a radius of two. When attacking a unit, the attacker's attack value is subtracted from the attacked unit's hit-point value. Units with hit-point value ≤ 0 are removed. The goal is to accumulate rewards by eliminating all opponents. The rewards are also closely following the default settings: 5 for killing an enemy unit, -0.1 for attacking an empty grid cell, 0.2 for attacking an enemy unit, and -1 for being killed (increased from the -0.1 default value). Additionally, there is a reward of 100 for killing all enemies. Each game is restricted to have a length of at most 300 time steps.

We use the full observability settings: there is no unit vision limitation or fog of war.

Necessarily, sector actions should translate into low-level actions for the agents located within. Here we choose to use 5 scripted algorithms to do so: four that move units in each of the cardinal directions and one for attacking enemy units, the same attacking script used by the enemy. In future work these scripts can be replaced with other fixed or even learnable agent policies, and the sector action can simply be which of these policies the agents within should use, similar to a meta-learning approach [66]. To avoid switching behaviors too often and to make exploration easier via temporal abstraction, the sector action is only changed every k environment time steps. A value of $k = 5$ was chosen based on a brief tuning process.

The network input is the global state, a 64×64 grid with 5 feature planes: obstacles, own and enemy units and own and enemy hit-points. All features are normalised by their maximum possible values.

10.4.2 Model and Training Settings

The network used for the sector abstraction is shown in Figure 10.1, with actions being emitted on a 4×4 sector granularity, unless otherwise specified. For IQL, the input is a 13×13 observation centered around the agent's position, with 7 feature planes. In addition to the 5 already mentioned, there are two more layers containing minimap information for both armies. For these layers, the original map unit information is scaled down to 13×13, and normalised such that each cell value is equal to the number of units within divided by total number of alive units. A value of 1 is added to the cell containing the acting agent, in the allied minimap. These minimap layers were included to provide the agents information about the global state.

The IQL network consists of one 4×4 stride-2 convolutional layer followed by two 3×3 stride-1 convolutional layers, all three with 32 filters and followed by ReLU activations and batch normalisation. A linear layer of 256 units follows, and two output heads for the value and advantage functions which are summed to extract the final Q-values.

The learning algorithm is based on DQN [150], with the dueling architecture update and N-step returns, with $N = 3$ across all experiments. The replay buffer contains the most recent 300k experiences. Training starts after the buffer is populated with 10k experiences. Batches of 128 experiences are sampled uniformly from the replay buffer every 128 steps played, and the network parameters are updated. The target network is updated every 3000 time steps. All experiments use $\gamma = 0.99$.

All networks are trained using the Adam learning algorithm [121] with the learning rate initialized with 0.000625 for our method and with 0.0001 for IQL, both chosen after brief tuning on a 40×40 scenario with 10 units per army. Exploration during training is performed using independent ε-greedy action selection, in which each sector chooses its action greedily using only its own Q_{S_i}. The ε value is annealed from 1.0 to 0.02 over the first 50k games and

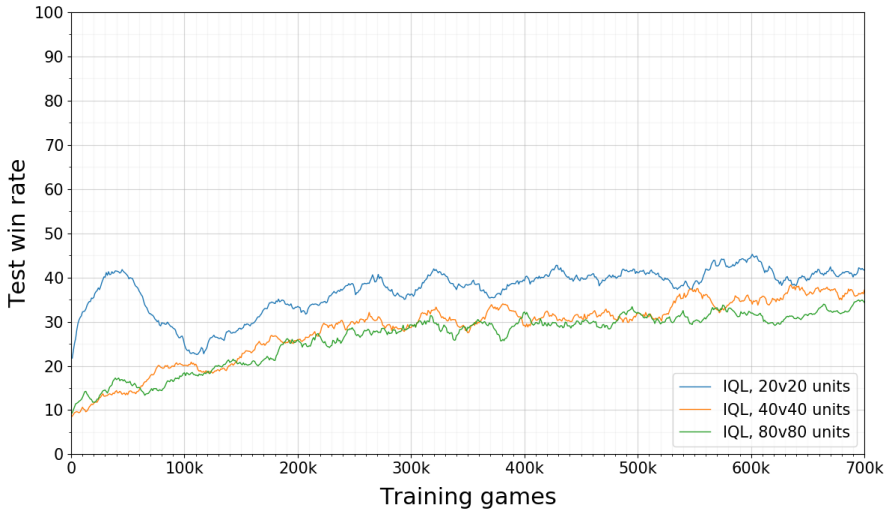


Figure 10.3: Test win rate for IQL and different army sizes measured every 1000 games during training, A sliding window of length 20 is used for smoothing

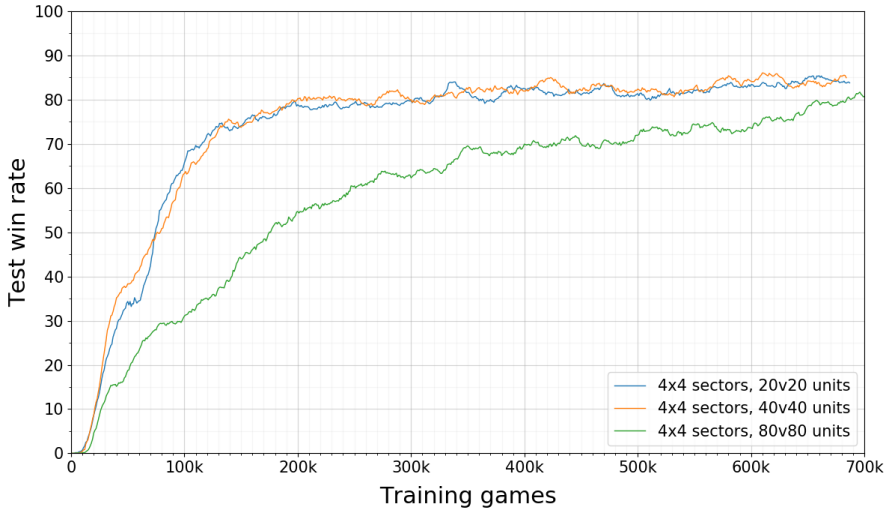


Figure 10.4: Test win rate for the 4×4 sector multi-agent action network and different army sizes. The same methodology was used as in Figure 10.3.

kept constant afterwards.

10.5 Results

We train all models with 700k games against the handcrafted script mentioned in the previous section. After every 1000 training games 100 test games are played independently with actions chosen greedily in evaluation mode (i.e., ε for exploration is set to 0). In what follows the proportion of these games in

which all enemy units are defeated within the target time limit is called *test win rate*.

Figures 10.3 and 10.4 show the test win rate for three single runs of IQL and Spatial Action Decomposition Learning, on 64×64 grid scenarios with 20, 40, and 80 units for each side. For the chosen parameter settings IQL fails to learn how to defeat the script consistently — not even achieving a 50% win rate. The training also appears to be unstable at times. As described in the introduction, this may be caused by all agents exploring simultaneously. We speculate that using VDNs could mitigate this problem, and isolating the effect of the sector abstraction, comparison to an IQL+VDN baseline is on our todo-list.

Our method shows a more stable learning behavior and can consistently defeat the opponent in more than 80% of the games after training. Controlling 80 units well seems significantly more difficult than 20 or 40, as shown by the slower learning process. We think this is because there are more units placed randomly behind the enemy lines. Grouping allied units while dealing with the enemy's requires more complex maneuvering, and often makes consolidation of one's army more difficult.

From a qualitative point of view, our method displays forms of coordination such as surrounding or attacking the enemy on a wider front and fleeing from a stronger group of enemies until regrouping with the rest of the army. Snapshots from a sample game are shown in Figure 10.7. We observed our system often using a small number of units as bait to split the enemy forces and fighting them one-by-one.

Learning with a model that uses 8×8 sectors is more challenging. Firstly, it gets stuck more often in local optima of predominantly choosing to attack

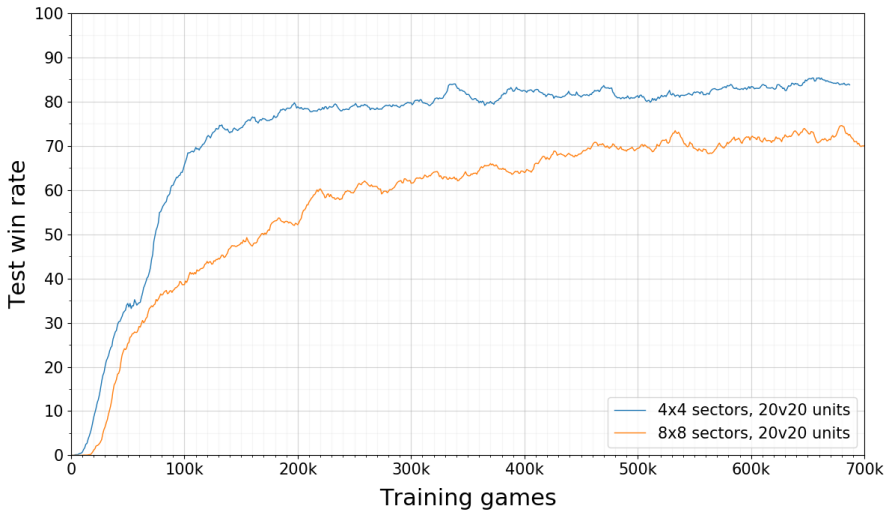


Figure 10.5: Test win rate for 4×4 and 8×8 sector decomposition, for models trained on scenarios with 20 units per army.

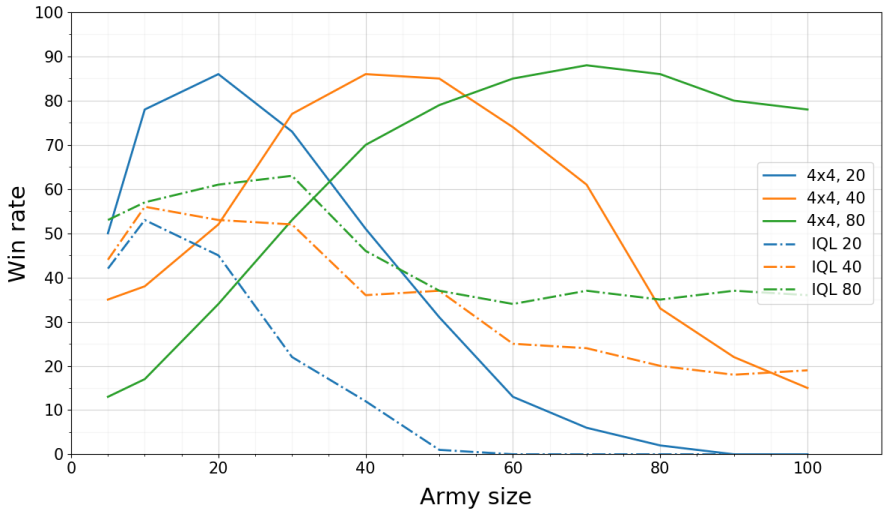


Figure 10.6: Results after $700k$ games of learning, for methods trained with scenarios of a specific army size (20, 40 or 80 units). Win rate is shown for evaluation in scenarios with a range of different army sizes, from 5 to 100.

and avoiding move actions. This is likely due to the more difficult exploration induced by the 64 sectors. Using mostly attack actions leads to test win rates around 50%, as expected. After all but the last network layer was initialised with weights from a trained 4×4 sectors model, learning proceeded smoothly and 75% win rates were reached, as seen in Figure 10.5. Secondly, exploring and assigning credit for four times as many sectors is more difficult and as such, learning converges more slowly with the maximum win rate being 10% lower.

Finally, Figure 10.6 shows the win rate of the 6 trained models from Figures 10.3 and 10.4 on scenarios with a range of different starting army sizes, instead of the only one used throughout the training. Models trained with fewer units do not generalize well, and both models that learned using a maximum of 20 units fail to win a single match when controlling more than 100 units. Interestingly, the sector models learn specialized tactics to solve the scenarios at hand, and the maximum win rate for a given army size is obtained by the model trained with that particular scenario. On the other hand, IQL is stronger across the board when training with more units and is best at controlling fewer units no matter the size of the training scenario.

10.6 Conclusions and Future Work

In this paper we study cooperative-competitive multi-agent RL with agents learning from a single team reward signal. Both individual as well as centralized learners fail to scale successfully with growing team sizes, and decomposing the joint action-value function into per-agent action-value functions has

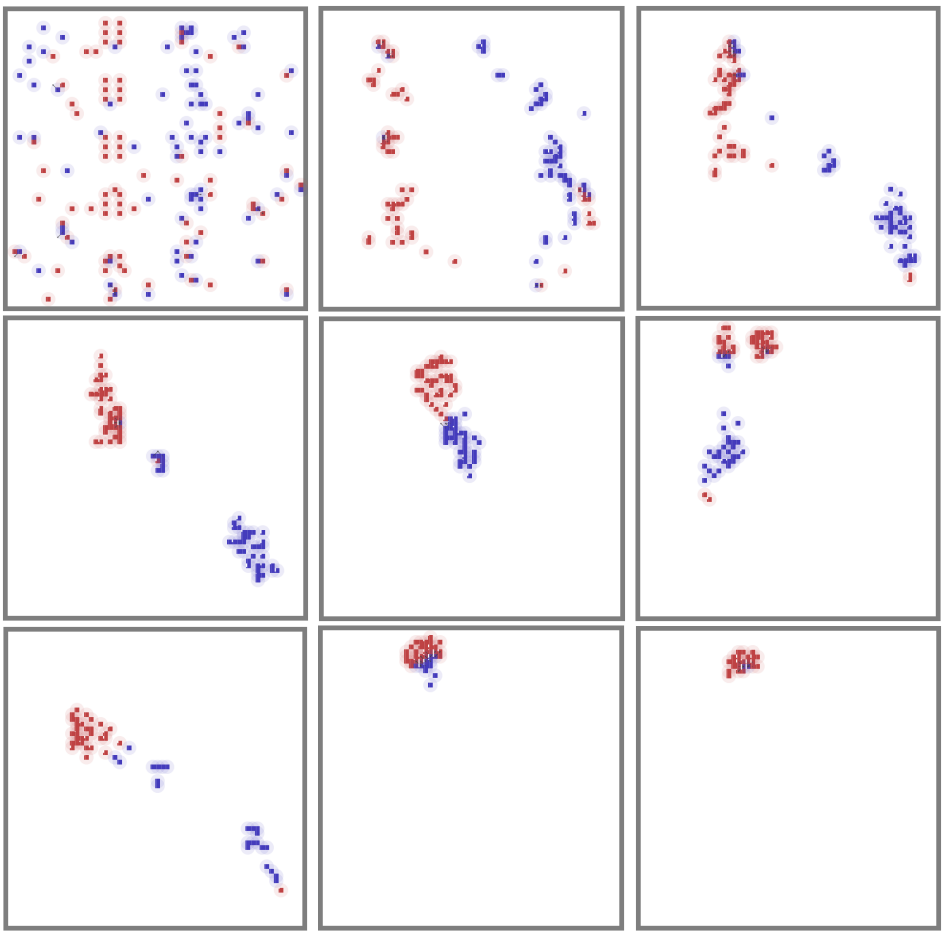


Figure 10.7: Snapshots from an 80v80 game played after 700k training games by the 4×4 sector algorithm (red units) against the scripted player (blue units). Baiting behavior to split the enemy forces can be observed multiple times.

previously shown great promise. Here we present a method that exploits existing spatial action correlations to decompose the joint action-value function on a per-sector basis instead, and show its effectiveness in both small and large scale scenarios. Results for combat scenarios with 20 to 80 units per side show improved performance over simple scripting and independent Q-learning.

Without a doubt the presented method can be improved in various ways. For instance, QMIX can be used to facilitate non-linear Q-value mixing, and low-level sector policies could be learned rather than using fixed scripts. Also, skip-connections can be used to enrich high-level features with low-level map details – providing high-fidelity context to lower-level action generation. Another interesting research avenue is a deeper integration of hierarchical action-value functions operating at different temporal scales [128], as temporally-abstracted exploration should help alleviate some of the problems that arise when decomposing the map in more and more sectors. Lastly, one can imagine

that based on the learned Q-functions it may be possible to focus Monte Carlo tree search similar to AlphaGo [195] to construct much stronger multi-agent systems.

10.7 Contributions Breakdown

This research was performed by Marius Stanescu, with supervision from Michael Buro. It was published just before the submission of this document, and there are no updates.

Chapter 11

Conclusions and Future Work

In this chapter we give an overview of the contributions made in this thesis, first as a summary then on a more detailed per-chapter basis and organized into three broad areas. We follow by a discussion of promising future topics for research.

11.1 Contributions

This thesis focuses on learning, using abstractions and adversarial search in real-time domains with large action and state spaces, such as RTS games. The presented abstract models for combat outcome prediction are very accurate while reasonably computationally inexpensive. Hence, they can be used in conjunction with higher level adversarial search algorithms to produce competitive agents. We introduced two approaches to designing such algorithms that are based on abstractions to reduce the search complexity. Firstly, Hierarchical Adversarial Search uses multiple search layers that work at different abstraction levels to decompose the original problem. Secondly, PuppetSearch methods use configurable scripts as an action abstraction mechanism and offer more flexibility. Both combat models and scripts have very basic spatial capabilities, if at all, and we showed how to use CNNs effectively to improve spatial awareness and increase the strength of search algorithms further.

Finally, for these complex domains full-game forward models rarely are readily available and require extra effort to design and adjust. End-to-end methods that learn or do not require implementation of such models are appealing. We have shown how the need of a forward model can be bypassed during gameplay by policy networks using convolutions, or eliminated altogether by using reinforcement learning algorithms. We obtained good results using reinforcement learning to train agents in small scale battle scenarios. However, these methods are challenged by larger environments, and learning becomes more difficult with increasing numbers of agents. We have introduced a new approach that uses CNNs to produce a spatial decomposition mechanism which learns action-value functions on a per-sector basis instead of

per-agent. Applied to a standard Q-learning method, this approach resulted in increased performance over the original algorithm in both small and large scale scenarios.

11.1.1 Combat Models

Chapter 3 describes a methodology for predicting the most probable army composition produced by an opponent in imperfect information strategy games. The presented prediction system is based on a declarative knowledge representation paradigm and thus background knowledge about game rules and player observations can easily be encoded for any given adversarial environment. The system uses the game rules knowledge and player observations to generate valid army compositions that the opponent might have over a given period of time. When combined with data extracted from human replays, this algorithm had very good prediction accuracy over intervals few minutes in length. The algorithm is able to provide these predictions in real-time and would be an useful component of high-level decision making algorithms in any adversarial domain with many rules that can be expressed as constraints and hidden opponent actions. In RTS games estimating the number of different types of units produced by an enemy can be used to inform build order and strategy selection, especially when coupled with the ability of predicting the outcome of such a possible encounter. Two combat models built for this purpose are described in the following two chapters.

Estimating outcomes for complex tactical problems is needed in high level strategic decisions, such as when to force or avoid fighting, timing a retreat or deciding what to build to best counter the opponent's forces. While we apply our research to RTS games, combat prediction could transfer to many other multi-player or multi-agent games of an adversarial nature such as team-based sports. Fast and accurate predictors are especially useful for look-ahead search algorithms such as the ones presented in Chapters 6 and 8: they can be used to assign agents to tasks, to estimate task completion likelihood or for state evaluation. In Chapter 4 we present a Bayesian model that can be used to predict battle outcomes, as well as predict what type and how many troops are needed to defeat a given army. Model parameters are learned from past encounters. Even training using only 10 battles, we achieve well over 80% prediction accuracy, better than any of the baseline algorithms do (logistic regression, naive bayes classifier, decision trees) even after seeing 400 battles. Moreover, it can suggest winning army compositions of equal size to the opponents with 87% accuracy. This was, to our knowledge, the first combat simulator applied to strategy games and trained on past encounters against specific opponents. We encountered some limitations when we tried incorporating it within our StarCraft AI bot, and we addressed them with subsequent research described below.

Chapter 5 introduces a model that generalized the Lanchester attrition

laws and successfully addressed the previous' research three main limitations:

- The previous model was linear in the unit features (i.e., the offensive score for a group of 10 marines is ten times the score for 1 marine). While accurate for close-ranged (melee) fights, it underestimated the effect of focusing fire in ranged fights.
- Only which army would win was predicted, but not the remaining army size. That means the old model could be used for state evaluation, but not as a forward model in look-ahead search algorithms. This is an important requirement for instances where we do not have access to the game forward model such as closed source commercial games.
- Experiments were run only using simulated data, without investigating the model's ability to adjust to different opponents by learning unit strength values from past battles.

Learning from the limitations of the first combat prediction algorithm, this new model fixed its most important deficits and proved to be quite useful in practice. Pitted against some of the best entries from a recent StarCraft AI competition, our bot with its simulation based attack-retreat code replaced by predictions made using the new algorithm showed encouraging performance gains. Overall, our combat model achieved better performance and was much faster than the original game engine or simulators such as SparCraft. This makes it suitable for search-based approaches that need a forward model to advance or predict the next state after executing a certain action. Indeed, when used as an evaluation function in research presented in Chapter 8, it led to stronger results compared to other alternative popular heuristics. We eventually introduced an even more accurate evaluation function method based on CNNs, which is more general and considers the entire game state instead of just combat units (Chapter 7).

11.1.2 Adversarial Search

As discussed in the introduction, standard adversarial tree search approaches are not directly applicable to problems with large state and action spaces. To reduce the search space, we propose different methods based on abstraction, and perform adversarial tree search in the smaller resulting sub-problems. Accurate evaluation functions like the ones described in the previous subsection also help by to obtain good results from shallower searches.

Chapter 6 describes Hierarchical Adversarial Search – an algorithm with multiple search layers that work at different abstraction levels. Hierarchical decomposition is used to provide the layers with partial or abstract views of the game state, and tasks to accomplish, in order to reduce the overall branching factor of the search. This approach benefits from fast and accurate predictors such as the ones described in Chapters 4 and 5 for estimating task

completion likelihood and unit assignment to tasks. A simulator was used as a basic forward model, and better results might have been obtained by replacing it with the combat model presented in Chapter 5. This method can be adapted to other adversarial problems with many agents, as long as they could be decomposed in relatively independent problems. However, despite promising results and potential, we discontinued this approach in favor of the subsequent PuppetSearch method (Chapter 8) due to difficulties in hand-crafting the multiple search layers and abstraction levels.

In Chapter 8 we present adversarial look-ahead algorithms that use configurable scripts with choice points as an action abstraction mechanism. We show that capable agents can be obtained even using basic PuppetSearch designs that select among a few given action scripts and that use coarsely abstracted simulators for world forwarding (Section 8.1). Different algorithm designs were explored in later research (Section 8.2) with the main results listed below:

- Using scripts with fewer choice points worked better on smaller maps, and conversely more choice points led to better performance on larger maps.
- Alpha-Beta variants performed better than MCTS ones, especially on larger maps.
- Running a deeper search by spreading the computation over multiple frames during which actions are generated according to the principal variation from the old search proved effective, especially on larger maps.

These script-based methods can be very valuable to commercial game AI designers, or to other domains where control over the range of resulting plans or behaviors is required. Techniques based on neural networks, for example, are sometimes less appealing when system decisions need to be justified and understood by designers. Configurable scripts can be used as an action abstraction mechanism in other domains, as long as they are combined with search and reasonably fast forward models or other methods to chose the next action – such as policy networks as in Section 8.3. Example applications might include airport routing and planning traffic lights, or power grid balancing.

Chapter 7 shows effective use of CNNs to learn the value of game state for RTS games. We showed significant improvement in accuracy over simpler state-of-the-art evaluations, including methods based on previous research described in Chapters 4 and 5 . The learned function evaluates the entire game state, and is not restricted to just tactical situations like the described combat models are. As a result, despite its much slower evaluation speed, on average the search algorithms that incorporated the CNN based evaluation function were considerably stronger. Even more, this method requires less domain knowledge and parameter tinkering compared to combat-specific models, and can be easily applied to many other grid-like domains.

We extended this research for a broader variety of scenarios and tested the new network design using competition size RTS maps in work described in Section 8.3. We also trained a policy version of the same network to predict the output of slower search-based algorithms. Using a policy network has several advantages, the first of which being that it eliminates the need of a forward model during gameplay. Secondly, it is much faster than the original search algorithm it was trained to emulate, which allowed the use of a separate tactical search algorithm to improve unit micromanagement. The resulting combined algorithm was stronger than all its individual components and other state-of-the-art algorithms. This gave us the confidence to use recent advances in neural network and reinforcement learning algorithms to bypass the need for forward models, which resulted in research summarized in the next subsection.

11.1.3 Deep Reinforcement Learning

Chapter 9 describes the application of popular RL algorithms such as DQN and A3C to learn how to fight battles in a AAA game without the use of simulators or forward models. The multi-agent learning task was decomposed into simultaneous single-agent problems via the IQL paradigm. To speed up learning in this complex environment with sparse rewards, we used techniques such as potential reward shaping and curriculum learning. The resulting agents demonstrated complex cooperative behaviors and soundly defeated the built-in game AI. In Section 9.4 we describe a hierarchical RL approach designed to scale better with the number of agents and to decrease the number of required games required for learning. A two-layer structure was used, with the higher level policy making decisions on a coarser layer of abstraction and choosing goals to be implemented by the lower level policy. While low level policies able to accomplish the given handcrafted objectives were obtained, training both high and low level policies jointly did not yield good results. Even so, we believe that learning hierarchically structured policies can improve sample efficiency, especially on new tasks (e.g., an unseen map or a scenario using a new type of unit) by re-using shared primitives executable over many time steps. These primitives can be policies that learn to accomplish a small set of fixed goals like in our case, or use a larger goal space, but they could also be much simpler, for example hand-crafted action scripts as in PuppetSearch. Temporal abstractions and shared policy hierarchies are proven to help on a wide range of environments with long time horizons, including 2D continuous movement, gridworld navigation, and 3D physics tasks [66]. Alternatively, in suitable domains spatial abstractions can be used to speed up learning, and research to this extent is described in the final chapter and summarized in the paragraph below.

Lastly, Chapter 10 describes a novel learning approach for cooperative-competitive multi-agent RL environments with agents learning from a single team reward signal. Existing methods are often limited to a small number of

agents, as learning becomes intractable when the agent number increases considerably. We described a method that overcomes inefficiencies of standard multi-agent Q-learning methods by exploiting existing spatial action correlations, which are leveraged by CNNs to decompose the joint action-value function on a per-sector basis. Fixed scripts are used as abstract actions chosen by each sector, but other options would also work because of the smaller dimensions of a sector compared to the full map (e.g., 1/16 or 1/64). Alternatives include algorithms discussed and used in previous chapters: search based algorithms if there is a forward model, or policies learnt from replays or via RL. Results for combat scenarios with 20 to 100 units per side showed improved performance over simple scripting and policies optimized by independent Q-learning. Our experiments demonstrated that our new architecture made long-term credit assignment from a single team reward more tractable, and improved scaling to larger numbers of agents. The proposed method could be applied to other environments that allow a hierarchical and spatial division of the space, for example 2D or 3D navigation or routing multi-agent tasks.

11.2 Future Work

We feel that one important area for future research is **model-based RL** [212]. Complex applications, such as AAA games, do not provide access to their internal forward models. Even if they do, they are usually too slow and use too many resources to be useful in online decision making, especially in look-ahead search algorithms. Handcrafting such a simulator is time consuming and the result is seldom as accurate as desired. The SparCraft simulator is one example: attempts at improving its movement simulation has shown that small StarCraft mechanics such as acceleration and turning animations which are not explicitly modeled by the simulator created cumulative errors that resulted in rapid divergence over time [188]. One solution is to use model-free RL to learn a Q-value function that estimates the expected reward of the different actions available in the present state, as we have done in Chapters 9 and 10. Model-free agents can greedily maximize their value function every time step, and do not require computationally expensive planning and look-ahead search. Another alternative which we propose for future work is to explicitly learn a model of the environment, which can be used for planning or training purposes. Recent research shows promising results of model-based agents in 3D shooter games. The *world models* [87] approach learns a compact model of the environment and can be trained relatively quickly in an unsupervised manner. The resulting world model can be used as a forward model, and agents can be trained entirely inside their own hallucinated dreams generated by it. Executing world model simulations can be done using GPUs in a distributed environment and would offer a significant speed-up compared to the original environment. Furthermore, the authors of [87] show that policies learnt only in the hallucinated world transfer surprisingly well to the actual,

real environment.

In Chapter 9 we suggested that in a multi-agent hierarchical framework, following high-level decisions for a fixed number of steps can increase the multi-modal aspect of the returns. Typical RL algorithms predict the average reward an agent receives from repeated attempts at a task, and use this estimation to choose the best action to take in a given state. However, randomness in the environment can produce outcomes very different from this average, and even orders with large positive expected values might lead to negative rewards. For example, defeating weak melee troops like a catapult squad is an easy task for a general's bodyguards in TotalWar games, but the one lucky shot they might pull off can happen to hit the general, causing morale wavering and significant impact on the battle. Randomness can certainly be part of the environment by design, it can arise because of partial observability and it will increase with the number of interacting agents. **Distributional RL** [17] is a paradigm that models not only the average but also the full variation of rewards, making it possible to differentiate between safe and risky choices that have similar expected returns. This can be valuable for games that require AIs to exhibit very stable combat behaviors and to spend their resources wisely, as well as for any other domain where budgeting plays an important part – for example money in online trading or time in traffic problems. Recent research introduced distributional generalizations of the DQN algorithm that are flexible, generally applicable and prove both faster to train and more accurate than previous models [48]. We believe that incorporating this paradigm would benefit learning in complex multi-agent domains, especially when large numbers of agents are learning simultaneously.

One of the issues that arose when designing RL agents for our research, most notably in Chapter 9, is that of alignment between the goals the agent is trying to achieve and those of its designer. Undesired behavior can emerge as a result misspecifying reward functions, which is a common cause of misalignment. For example, agents in a boat racing game learnt to completely ignore desired behavior of racing and to repeatedly collect points instead by moving in a small loop [45]. Agents can even prefer to get killed on purpose at the end of one level, to avoid having to play a more difficult second level [183]. In our case, we found that the process of manually redesigning the reward functions to encourage some behaviors and discourage others is a tedious process, and we believe that approaches that optimize a learned reward function are promising avenues to be explored in this context. For example, recent work defines desired goals using non-expert human preferences between pairs of trajectory segments [36]. This approach has been shown to solve complex RL tasks (including Atari games) effectively without access to the reward function, with human feedback on less than one percent of the agent's interactions. We believe this technique would reduce the need of anticipating unexpected scenarios and behaviors while designing the agent, by adding new training data to deal with issues as they are discovered.

Another interesting research avenue lies in the direction of hierarchical RL. Very recent research shows promising results using a two-level hierarchical RL algorithm built on top of 165 abstract macro actions that were designed manually [210]. While the resulting agent consistently defeats the built-in StarCraft II AI and above average-level human players for the first time in StarCraft games, it has lost against expert (top 50% - 30%) human players. Inspired by these results, we continue exploring hierarchical RL ideas which require less hand-crafting effort. Firstly, future work should explore replacing the scripts used to generate low-level actions the method presented in Chapter 10 with learnable policies. Secondly, the approaches presented in Chapters 6, 8, 9 and 10 only possess two temporal scales. While the PuppetSearch action scripts and HAS tasks are abstract enough primitives, in the RL experiments presented in Chapters 9 and 10 we used a fixed ratio of 10 and 5 real-world steps respectively to one high-level action. A deeper or more temporally abstracted architecture should help with exploration and long horizon tasks. Finally, we note that designing task-specific goals to be used in conjunction with temporal abstraction requires effort and might not generalize well between tasks. In future work we would like to study proposing and learning goals automatically by the high-level controllers, which has been shown to work even with off-policy experience based methods [155]. The authors introduce off-policy corrections to alleviate the fact that low-level behaviors change the action space for the high-level policies, and learn goals in the same space as the problem's state space resulting in a generally applicable and very sample-efficient algorithm.

There is on-going research in several other adjacent areas that could offer insights and benefit our work. To mention a few, *transfer learning* is a popular approach where models developed for a task are reused as starting points for models on other tasks, where it allows rapid progress or improved performance. Using simpler scenarios in which agents can master a set of specific skills and then transferring knowledge to new environments can enable learning and drastically reduce the learning time on very challenging tasks. Another relevant problem is *few shot learning*, which deals with the lack of training data (e.g., we might only have a few human expert game replays) and an algorithm's ability to adapt to unforeseen situations that have not been encountered during training. Using standard techniques in these scenarios often leads to overfitting the data, and the model has to be forced to generalize beyond the few available training instances.

Lastly, further effort at encouraging the videogame industry to adopt techniques described in this dissertation should be undertaken. We have participated in many AI competitions for RTS games, and while we are far from defeating human players as in Go [195] or more recently Dota II [167], these efforts showcase the benefits over more traditional AI methods. We have published tutorial versions of Chapters 5 and 8 in a book directed at industry professionals [203, 12], and the availability of a new generation of open-source tools that empower game developers with ML techniques [225] will make such

efforts easier in the future. Finally, we feel that efforts at bringing the games industry and AI research closer, such as the internship presented in Chapter 9, are a positive step in this direction, and benefit both parties.

Bibliography

[1] Deep Thought project. Available at http://deepthought.23inch.de/. Accessed: 2013-08-30.

[2] Spring engine. `https://springrts.com/`, (accessed September 25, 2018).

[3] Wargus. `https://wargus.github.io/`, (accessed September 25, 2018).

[4] Samuel Alvernaz and Julian Togelius. Autoencoder-augmented neuroevolution for visual doom playing. In *Computational Intelligence and Games (CIG), 2017 IEEE Conference on*, pages 1–8. IEEE, 2017.

[5] Leonardo R Amado and Felipe Meneguzzi. Reinforcement learning applied to rts games. *AAMAS Workshop: Adaptive Learning Agents*, 2017.

[6] Pierre-Luc Bacon, Jean Harb, and Doina Precup. The option-critic architecture. In *AAAI*, pages 1726–1734, 2017.

[7] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[8] Chitta Baral. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press, 2003.

[9] Nicolas A. Barriga, Marius Stanescu, and Michael Buro. Building placement optimization in real-time strategy games. In *Workshop on Artificial Intelligence in Adversarial Real-Time Games, AIIDE*, 2014.

[10] Nicolas A. Barriga, Marius Stanescu, and Michael Buro. Parallel UCT search on GPUs. In *IEEE Conference on Computational Intelligence and Games (CIG)*, 2014.

[11] Nicolas A. Barriga, Marius Stanescu, and Michael Buro. Puppet Search: Enhancing scripted behaviour by look-ahead search with applications to Real-Time Strategy games. In *Eleventh Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, pages 9–15, 2015.

[12] Nicolas A. Barriga, Marius Stanescu, and Michael Buro. Combining scripted behavior with game tree search for stronger, more robust game AI. In *Game AI Pro 3: Collected Wisdom of Game AI Professionals*, chapter 14. CRC Press, 2017.

[13] Nicolas A. Barriga, Marius Stanescu, and Michael Buro. Combining strategic learning and tactical search in Real-Time Strategy games. In *Accepted for presentation at the Thirteenth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 2017.

[14] Nicolas A. Barriga, Marius Stanescu, and Michael Buro. Game tree search based on non-deterministic action scripts in real-time strategy games. *IEEE Transactions on Computational Intelligence and AI in Games (TCIAIG)*, 2017.

[15] Gabriel Barth-Maron, Matthew W Hoffman, David Budden, Will Dabney, Dan Horgan, Alistair Muldal, Nicolas Heess, and Timothy Lillicrap. Distributed distributional deterministic policy gradients. *arXiv preprint arXiv:1804.08617*, 2018.

[16] James Batchelor. Games industry generated $108.4bn in revenues in 2017. `https://www.gamesindustry.biz/articles/2018-01-31-games-industry-generated-usd108-4bn-in-revenues-in-2017`, 2018 (accessed September 25, 2018).

[17] Marc G Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. In *International Conference on Machine Learning*, pages 449–458, 2017.

[18] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[19] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48. ACM, 2009.

[20] Alex Bernstein and M de V Roberts. Computer v chess-player. *Scientific American*, 198(6):96–105, 1958.

[21] Blizzard Entertainment. StarCraft: Brood War. http://us.blizzard.com/en-us/games/sc/, 1998.

[22] Miroslav Bogdanovic, Dejan Markovikj, Misha Denil, and Nando De Freitas. Deep apprenticeship learning for playing video games. In *AAAI Workshop: Learning for General Competency in Video Games*, 2015.

[23] Michael Bowling, Neil Burch, Michael Johanson, and Oskari Tammelin. Heads-up limit hold'em poker is solved. *Science*, 347(6218):145–149, 2015.

[24] Louis Brandy. Evolution chamber: Using genetic algorithms to find StarCraft 2 build orders. http://lbrandy.com/blog/2010/11/using-genetic-algorithms-to-find-starcraft-2-build-orders/, November 2010.

[25] Wray L. Buntine. Operations for learning with graphical models. *arXiv preprint cs/9412102*, 1994.

[26] Michael Buro. ORTS: A hack-free RTS game environment. In *International Conference on Computers and Games*, pages 280–291. Springer, 2002.

[27] Michael Buro. Real-time strategy games: A new AI research challenge. In *IJCAI 2003*, pages 1534–1535. International Joint Conferences on Artificial Intelligence, 2003.

[28] Michael Buro. Call for AI research in RTS games. In *Proceedings of the AAAI-04 Workshop on Challenges in Game AI*, pages 139–142, 2004.

[29] Michael Buro. ORTS competitions. `https://skatgame.net/mburo/orts/index.html#Competitions`, 2009 (accessed September 25, 2018).

[30] Michael Buro and David Churchill. Real-time strategy game competitions. *AI Magazine*, 33(3):106–108, 2012.

[31] Lucian Busoniu, Robert Babuska, and Bart De Schutter. A comprehensive survey of multiagent reinforcement learning. *IEEE Trans. Systems, Man, and Cybernetics, Part C*, 38(2):156–172, 2008.

[32] Murray Campbell, A Joseph Hoane, and Feng-hsiung Hsu. Deep Blue. *Artificial intelligence*, 134(1):57–83, 2002.

[33] Martin Čertickỳ and Michal Čertickỳ. Case-based reasoning for army compositions in real-time strategy games. In *Proceedings of Scientific Conference of Young Researchers*, pages 70–73, 2013.

[34] Michal Čertickỳ. Implementing a wall-in building placement in StarCraft with declarative programming. *arXiv preprint arXiv:1306.4460*, 2013.

[35] Esports Charts. 2017 Esports charts results. `https://esc.watch/blog/post/results-2017`, 2017 (accessed September 25, 2018).

[36] Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. In *Advances in Neural Information Processing Systems*, pages 4299–4307, 2017.

[37] Michael Chung, Michael Buro, and Jonathan Schaeffer. Monte Carlo planning in RTS games. In *IEEE Symposium on Computational Intelligence and Games (CIG)*, 2005.

[38] David Churchill. SparCraft: open source StarCraft combat simulation. http://code.google.com/p/sparcraft/, 2013.

[39] David Churchill. A history of starcraft AI competitions. `https://www.cs.mun.ca/~dchurchill/starcraftaicomp/history.shtml`, 2016 (accessed September 25, 2018).

[40] David Churchill and Michael Buro. Build order optimization in StarCraft. In *AI and Interactive Digital Entertainment Conference, AIIDE (AAAI)*, pages 14–19, 2011.

[41] David Churchill and Michael Buro. Incorporating search algorithms into RTS game agents. In *AIIDE Workshop on Artificial Intelligence in Adversarial Real-Time Games*, 2012.

[42] David Churchill and Michael Buro. Incorporating search algorithms into RTS game agents. In *AI and Interactive Digital Entertainment Conference, AIIDE (AAAI)*, 2012.

[43] David Churchill and Michael Buro. Portfolio greedy search and simulation for large-scale combat in StarCraft. In *IEEE Conference on Computational Intelligence in Games (CIG)*, pages 1–8. IEEE, 2013.

[44] David Churchill, Abdallah Saffidine, and Michael Buro. Fast heuristic search for RTS game combat scenarios. In *Proceedings of the Eighth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, AIIDE'12, pages 112–117, 2012.

[45] Jack Clark and Dario Amodei. Faulty reward functions in the wild. `https://blog.openai.com/faulty-reward-functions/`, 2016 (accessed September 25, 2018).

[46] C Claus and C Boutilier. The dynamics of reinforcement learning in cooperative multiagent systems. *AAAI*, pages 746–752, 1998.

[47] Mitchell K Colby, Sepideh Kharaghani, Chris HolmesParker, and Kagan Tumer. Counterfactual exploration for improving multiagent learning. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, pages 171–179. International Foundation for Autonomous Agents and Multiagent Systems, 2015.

[48] Will Dabney, Georg Ostrovski, David Silver, and Rémi Munos. Implicit quantile networks for distributional reinforcement learning. 2018.

[49] Holger Danielsiek, Raphael Stuer, Andreas Thom, Nicola Beume, Boris Naujoks, and Mike Preuss. Intelligent moving of groups in real-time strategy games. *2008 IEEE Symposium On Computational Intelligence and Games*, pages 71–78, 2008.

[50] Peter Dayan and Geoffrey E Hinton. Feudal reinforcement learning. In S. J. Hanson, J. D. Cowan, and C. L. Giles, editors, *Advances in Neural Information Processing Systems 5*, pages 271–278. Morgan-Kaufmann, 1993.

[51] Ethan Dereszynski, Jesse Hostetler, Alan Fern, Tom Dietterich Thao-Trang Hoang, and Mark Udarbe. Learning probabilistic behavior models in real-time strategy games. In AAAI, editor, *Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 2011.

[52] Sam Devlin, Logan Yliniemi, Daniel Kudenko, and Kagan Tumer. Potential-based difference rewards for multiagent reinforcement learning. In *Proceedings of the 2014 international conference on Autonomous agents and multi-agent systems*, pages 165–172. International Foundation for Autonomous Agents and Multiagent Systems, 2014.

[53] Chao Dong, Chen Change Loy, Kaiming He, and Xiaoou Tang. Learning a deep convolutional network for image super-resolution. In *Computer Vision–ECCV 2014*, pages 184–199. Springer, 2014.

[54] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. mar 2016.

[55] Steve Dworschak, Susanne Grell, Victoria J Nikiforova, Torsten Schaub, and Joachim Selbig. Modeling biological networks by action languages via Answer Set Programming. *Constraints*, 13(1-2):21–65, 2008.

[56] e-Sports Earnings. Overall eSports stats and top games of 2017. `https://www.esportsearnings.com/history/2017/games`, 2017 (accessed September 25, 2018).

[57] Adam Eck, Leen-Kiat Soh, Sam Devlin, and Daniel Kudenko. Potential-based reward shaping for finite horizon online pomdp planning. *Autonomous Agents and Multi-Agent Systems*, 30(3):403–445, 2016.

[58] Arpad E Elo. *The rating of chessplayers, past and present*, volume 3. Batsford London, 1978.

[59] Graham Erickson and Michael Buro. Global state evaluation in StarCraft. In *Proceedings of the Tenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, pages 112–118. AAAI Press, 2014.

[60] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymir Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. *arXiv preprint arXiv:1802.01561*, 2018.

[61] Facebook. Facebook AI research. `https://research.fb.com/category/facebook-ai-research/`, 2018 (accessed September 25, 2018).

[62] Jakob Foerster, Gregory Farquhar, Triantafyllos Afouras, Nantas Nardelli, and Shimon Whiteson. Counterfactual Multi-Agent policy gradients. May 2017.

[63] Jakob Foerster, Nantas Nardelli, Gregory Farquhar, Philip H S Torr, Pushmeet Kohli, and Shimon Whiteson. Stabilising experience replay for deep Multi-Agent reinforcement learning. In *Thirty-fourth International Conference on Machine Learning*, 2017.

[64] Kenneth D. Forbus, James V. Mahoney, and Kevin Dill. How qualitative spatial reasoning can improve strategy game AIs. *IEEE Intelligent Systems*, 17:25–30, July 2002.

[65] Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, et al. Noisy networks for exploration. *arXiv preprint arXiv:1706.10295*, 2017.

[66] Kevin Frans, Jonathan Ho, Xi Chen, Pieter Abbeel, and John Schulman. Meta learning shared hierarchies. October 2017.

[67] Scott Fujimoto, Herke van Hoof, and Dave Meger. Addressing function approximation error in actor-critic methods. *arXiv preprint arXiv:1802.09477*, 2018.

[68] Timothy Furtak and Michael Buro. On the complexity of two-player attrition games played on graphs. In G. Michael Youngblood and Vadim Bulitko, editors, *Proceedings of the Sixth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2010*, Stanford, California, USA, October 2010.

[69] Johannes Fürnkranz and Miroslav Kubat. *Machines that learn to play games*. Nova Publishers, 2001.

[70] Martin Gebser, Torsten Grote, Roland Kaminski, and Torsten Schaub. Reactive answer set programming. In *Proceedings of Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 54–66. Springer, 2011.

[71] Martin Gebser, Carito Guziolowski, Mihail Ivanchev, Torsten Schaub, Anne Siegel, Sven Thiele, and Philippe Veber. Repair and prediction (under inconsistency) in large biological networks with Answer Set Programming. In *Proceedings of the Twelfth International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 497–507, 2010.

[72] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. A user's guide to gringo, clasp, clingo, and iclingo, 2008. Available at: http://potassco.sourceforge.net/.

[73] Christopher W. Geib and Robert P. Goldman. A probabilistic plan recognition algorithm based on plan tree grammars. *Artificial Intelligence*, 173:1101–1132, July 2009.

[74] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Logic Programming: Proceedings of Fifth International Conference and Symposium*, volume 88, pages 1070–1080. MIT Press, Cambridge MA, 1988.

[75] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New generation computing*, 9(3-4):365–385, 1991.

[76] Elizabeth Gibney. Self-taught AI is best yet at strategy game Go. *Nature News*, 550:16–7, 2017.

[77] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014.

[78] Mark E Glickman. Parameter estimation in large dynamic paired comparison experiments. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 48(3):377–394, 1999.

[79] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *International conference on artificial intelligence and statistics*, pages 249–256, 2010.

[80] Google. Planet Wars AI challenge. `https://web.archive.org/web/20120119040848/http://planetwars.aichallenge.org/`, 2010 (accessed September 25, 2018).

[81] Google. Ants AI challenge. `http://ants.aichallenge.org/`, 2011 (accessed September 25, 2018).

[82] Google. DeepMind. `https://deepmind.com/`, accessed September 25, 2018.

[83] Shixiang Gu, Timothy Lillicrap, Zoubin Ghahramani, Richard E Turner, and Sergey Levine. Q-prop: Sample-efficient policy gradient with an off-policy critic. *arXiv preprint arXiv:1611.02247*, 2016.

[84] Carlos Guestrin, Daphne Koller, and Ronald Parr. Multiagent planning with factored MDPs. In *Advances in neural information processing systems*, pages 1523–1530, 2002.

[85] Xiaoxiao Guo, Satinder Singh, Honglak Lee, Richard L Lewis, and Xiaoshi Wang. Deep learning for real-time atari game play using offline monte-carlo tree search planning. In *Advances in Neural Information Processing Systems*, pages 3338–3346, 2014.

[86] Jayesh K Gupta, Maxim Egorov, and Mykel Kochenderfer. Cooperative multi-agent control using deep reinforcement learning. In *Autonomous Agents and Multiagent Systems*, pages 66–83. Springer International Publishing, 2017.

[87] David Ha and Jurgen Schmidhuber. World models. *arXiv preprint arXiv:1803.10122*, March 2018.

[88] Johan Hagelbäck and Stefan J. Johansson. Dealing with fog of war in a real time strategy game environment. In *Computational Intelligence and Games, 2008. CIG'08. IEEE Symposium On*, pages 55–62. IEEE, 2008.

[89] Johan Hagelbäck and Stefan J. Johansson. A multiagent potential field-based bot for real-time strategy games. *International Journal of Computer Games Technology*, 2009:4:1–4:10, January 2009.

[90] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.

[91] Jun Han and Claudio Moraga. The influence of the sigmoid function parameters on the speed of backpropagation learning. In *From Natural to Artificial Neural Computation*, pages 195–201. Springer, 1995.

[92] Byron R Harder, Imre Balogh, and CJ Darken. Implementation of an automated fire support planner. In *Proceedings of the Twelfth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-16)*, 2016.

[93] Hado V Hasselt. Double q-learning. In *Advances in Neural Information Processing Systems*, pages 2613–2621, 2010.

[94] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, pages 2094–2100. AAAI Press, 2016.

[95] Matthew Hausknecht, Joel Lehman, Risto Miikkulainen, and Peter Stone. A neuroevolution approach to general atari game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(4):355–366, 2014.

[96] Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. In *2015 AAAI Fall Symposium Series*, 2015.

[97] Matthew John Hausknecht. *Cooperation and communication in multiagent deep reinforcement learning*. PhD thesis, The University of Texas at Austin, 2016.

[98] He He, UMD EDU, Jordan Boyd-Graber, and Hal Daumé III. Opponent modeling in deep reinforcement learning. In *Proceedings of The 33rd International Conference on Machine Learning*, pages 1804–1813, 2016.

[99] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

[100] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. pages 1026–1034, 2015.

[101] Nicolas Heess, Greg Wayne, Yuval Tassa, Timothy Lillicrap, Martin Riedmiller, and David Silver. Learning and transfer of modulated locomotor controllers. *arXiv preprint arXiv:1610.05182*, 2016.

[102] Adam Heinermann. Broodwar API. http://code.google.com/p/bwapi/, 2014.

[103] Johannes Heinrich and David Silver. Deep reinforcement learning from self-play in imperfect-information games. *arXiv preprint arXiv:1603.01121*, 2016.

[104] Ralf Herbrich, Tom Minka, and Thore Graepel. Trueskill ™: A Bayesian skill rating system. *Advances in Neural Information Processing Systems*, 19:569, 2007.

[105] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. *arXiv preprint arXiv:1710.02298*, 2017.

[106] Stephen Hladky and Vadim Bulitko. An evaluation of models for predicting opponent positions in first-person shooter video games. In *CIG (IEEE)*, 2008.

[107] Jesse Hostetler, Ethan W Dereszynski, Thomas G Dietterich, and Alan Fern. Inferring strategies from limited reconnaissance in real-time strategy games. *arXiv preprint arXiv:1210.4880*, 2012.

[108] Ryan Houlette and Dan Fu. The ultimate guide to FSMs in games. *AI Game Programming Wisdom 2*, 2003.

[109] Ji-Lung Hsieh and Chuen-Tsai Sun. Building a player strategy model by analyzing replays of real-time strategy games. In *IJCNN*, pages 3106–3111, 2008.

[110] U. Jaidee and H. Muñoz-Avila. CLASSQ-L: A Q-learning algorithm for adversarial real-time strategy games. In *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2012.

[111] Ulit Jaidee, Héctor Muñoz-Avila, and David W Aha. Case-based learning in goal-driven autonomy agents for real-time strategy combat tasks. In *Proceedings of the ICCBR Workshop on Computer Games*, pages 43–52, 2011.

[112] E. T. Jaynes. *Probability Theory: The Logic of Science*. Cambridge University Press, June 2003.

[113] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[114] Rie Johnson and Tong Zhang. Effective use of word order for text categorization with convolutional neural networks. *arXiv preprint arXiv:1412.1058*, 2014.

[115] Emilio Jorge, Mikael Kågebäck, Fredrik D Johansson, and Emil Gustavsson. Learning to play guess who? and inventing a grounded language as a consequence. November 2016.

[116] Niels Justesen, Philip Bontrager, Julian Togelius, and Sebastian Risi. Deep learning for video game playing. *arXiv preprint arXiv:1708.07902*, 2017.

[117] Niels Justesen and Sebastian Risi. Learning macromanagement in starcraft from replays using deep learning. *arXiv preprint arXiv:1707.03743*, 2017.

[118] Froduald Kabanza, Philipe Bellefeuille, Francis Bisson, Abder Rezak Benaskeur, and Hengameh Irandoust. Opponent behaviour recognition for real-time strategy games. In *AAAI Workshops*, 2010.

[119] Łukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. *arXiv preprint arXiv:1511.08228*, 2015.

[120] Alex Kendall, Matthew Grimes, and Roberto Cipolla. Posenet: A convolutional network for real-time 6-dof camera relocalization. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2938–2946, 2015.

[121] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[122] Daphne Koller and Nir Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[123] George Konidaris and Andrew G Barto. Building portable options: Skill transfer in reinforcement learning. In *IJCAI*, volume 7, pages 895–900, 2007.

[124] Harald Köstler and Björn Gmeiner. A multi-objective genetic algorithm for build order optimization in StarCraft II. *KI-Künstliche Intelligenz*, 27(3):221–233, 2013.

[125] Alex Kovarsky and Michael Buro. A first look at build-order optimization in real-time strategy games. In *Proceedings of the GameOn Conference*, pages 18–22, 2006.

[126] Alexander Kovarsky and Michael Buro. Heuristic search applied to abstract combat games. *Advances in Artificial Intelligence*, pages 66–78, 2005.

[127] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[128] Tejas D Kulkarni, Karthik Narasimhan, Ardavan Saeedi, and Josh Tenenbaum. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. In *Advances in neural information processing systems*, pages 3675–3683, 2016.

[129] Jasper Laagland. A htn planner for a real-time strategy game. *Unpublished manuscript. (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.406.8722&rep=rep1&type=pdf)*, 2008.

[130] John Laird and Michael VanLent. Human-level ai's killer application: Interactive computer games. *AI magazine*, 22(2):15, 2001.

[131] Guillaume Lample and Devendra Singh Chaplot. Playing fps games with deep reinforcement learning. In *AAAI*, pages 2140–2146, 2017.

[132] Frederick William Lanchester. *Aircraft in warfare: The dawn of the fourth arm*. Constable limited, 1916.

[133] Guillaume J Laurent, Laëtitia Matignon, Le Fort-Piat, and Others. The world of independent learners is not Markovian. *International Journal of Knowledge-based and Intelligent Engineering Systems*, 15(1):55–64, 2011.

[134] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[135] Yann LeCun and Marc Aurelio Ranzato. Deep learning. Tutorial at ICML, 2013.

[136] Joel Z Leibo, Vinicius Zambaldi, Marc Lanctot, Janusz Marecki, and Thore Graepel. Multi-agent reinforcement learning in sequential social dilemmas. February 2017.

[137] Andrew Levy, Robert Platt, and Kate Saenko. Hierarchical actor-critic. *arXiv preprint arXiv:1712.00948*, 2017.

[138] V Lisỳ, B Bošanskỳ, R Vaculín, and Michal Pechoucek. Agent subset adversarial search for complex non-cooperative domains. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, pages 211–218. IEEE, 2010.

[139] Viliam Lisỳ, Branislav Bošanskỳ, Michal Jakob, and Michal Pechoucek. Adversarial search with procedural knowledge heuristic. In *Proc. of 8th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2009)*, 2009.

[140] Liang Liu and Longshu Li. Regional cooperative multi-agent Q-learning based on potential field. pages 535–539, 2008.

[141] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3431–3440, 2015.

[142] John Manslow. Using reinforcement learning to solve ai control problems. *AI Game Programming Wisdom*, 2:591–601, 2004.

[143] Bhaskara Marthi, Stuart Russell, David Latham, and Carlos Guestrin. Concurrent hierarchical reinforcement learning. In *International Joint Conference of Artificial Intelligence, IJCAI*, pages 779–785, 2005.

[144] Alexandre Menif, Christophe Guettier, and Tristan Cazenave. Planning and execution control architecture for infantry serious gaming. In *Planning in Games Workshop*, page 31, 2013.

[145] Microsoft. Microsoft research lab – Cambridge. `https://www.microsoft.com/en-us/research/lab/microsoft-research-cambridge/`, accessed September 25, 2018.

[146] Thomas P Minka. *A family of algorithms for approximate Bayesian inference*. PhD thesis, Massachusetts Institute of Technology, 2001.

[147] Kinshuk Mishra, Santiago Ontañón, and Ashwin Ram. Situation assessment for plan retrieval in real-time strategy games. In *ECCBR*, pages 355–369, 2008.

[148] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[149] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. In *NIPS Deep Learning Workshop*. 2013.

[150] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[151] Kenrick J Mock. Hierarchical heuristic search techniques for empire-based games. In *Proceedings of the International Conference on Artificial Intelligence (IC-AI)*, pages 643–648, 2002.

[152] Matt Molineaux, Matthew Klenk, and David W Aha. Goal-driven autonomy in a navy strategy simulation. Technical report, DTIC Document, 2010.

[153] Maximilian Möller, Marius Schneider, Martin Wegner, and Torsten Schaub. Centurio, a general game player: Parallel, Java- and ASP-based. *KI-Künstliche Intelligenz*, 25(1):17–24, 2011.

[154] Dov Monderer and Lloyd S Shapley. Fictitious play property for games with identical interests. *Journal of economic theory*, 68(1):258–265, 1996.

[155] Ofir Nachum, Shane Gu, Honglak Lee, and Sergey Levine. Data-efficient hierarchical reinforcement learning. *arXiv preprint arXiv:1805.08296*, 2018.

[156] Ofir Nachum, Mohammad Norouzi, Kelvin Xu, and Dale Schuurmans. Trust-pcl: An off-policy trust region method for continuous control. *arXiv preprint arXiv:1707.01891*, 2017.

[157] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, et al. Massively parallel methods for deep reinforcement learning. *arXiv preprint arXiv:1507.04296*, 2015.

[158] Negin Nejati, Pat Langley, and Tolga Konik. Learning hierarchical task networks by observation. In *Proceedings of the 23rd international conference on Machine learning*, pages 665–672. ACM, 2006.

[159] Santi Ontañón, Kinshuk Mishra, Neha Sugandh, and Ashwin Ram. On-line case-based planning. *Computational Intelligence*, 26(1):84–119, 2010.

[160] Santiago Ontañón. Experiments with game tree search in real-time strategy games. *arXiv preprint arXiv:1208.1940*, 2012.

[161] Santiago Ontañón. The combinatorial multi-armed bandit problem and its application to real-time strategy games. In *AIIDE*, 2013.

[162] Santiago Ontañón. Informed monte carlo tree search for real-time strategy games. In *Computational Intelligence and Games (CIG), 2016 IEEE Conference on*, pages 1–8. IEEE, 2016.

[163] Santiago Ontañón, Nicolas A Barriga, Cleyton R Silva, Rubens O Moraes, and Levi HS Lelis. The first microrts artificial intelligence competition. *AI Magazine*, 39(1), 2018.

[164] Santiago Ontañón and Michael Buro. Adversarial hierarchical-task network planning for complex real-time games. In *Proceedings of the 24th International Conference on Artificial Intelligence (IJCAI)*, pages 1652–1658, 2015.

[165] Santiago Ontañón, Kinshuk Mishra, Neha Sugandh, and Ashwin Ram. Case-based planning and execution for real-time strategy games. In *ICCBR '07*, pages 164–178, Berlin, Heidelberg, 2007. Springer-Verlag.

[166] Santiago Ontañón, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill, and Mike Preuss. A survey of real-time strategy game AI research and competition in StarCraft. *TCIAIG*, 5(4):293–311, 2013.

[167] OpenAI. The International 2018: Results. `https://blog.openai.com/the-international-2018-results/`, 2018 (accessed September 25, 2018).

[168] OpenAI. OpenAI Five. `https://blog.openai.com/openai-five/`, 2018 (accessed September 25, 2018).

[169] Liviu Panait and Sean Luke. Cooperative multi-agent learning: The state of the art. *Auton. Agent. Multi. Agent. Syst.*, 11(3):387–434, November 2005.

[170] Zhen-Jia Pang, Ruo-Ze Liu, Zhou-Yu Meng, Yi Zhang, Yang Yu, and Tong Lu. On reinforcement learning for full-length game of starcraft. *arXiv preprint arXiv:1809.09095*, 2018.

[171] Peng Peng, Quan Yuan, Ying Wen, Yaodong Yang, Zhenkun Tang, Haitao Long, and Jun Wang. Multiagent Bidirectionally-Coordinated nets for learning to play StarCraft combat games. 29 March 2017.

[172] Luke Perkins. Terrain analysis in real-time strategy games: An integrated approach to choke point detection and region decomposition. *Artificial Intelligence*, pages 168–173, 2010.

[173] Mike Preuss, Nicola Beume, Holger Danielsiek, Tobias Hein, Boris Naujoks, Nico Piatkowski, Raphael Stuer, Andreas Thom, and Simon Wessing. Towards intelligent team composition and maneuvering in real-time strategy games. *IEEE Transactions on Computational Intelligence and AI in Games*, (2):82–98, 2010.

[174] Tabish Rashid, Mikayel Samvelyan, Christian Schroeder de Witt, Gregory Farquhar, Jakob Foerster, and Shimon Whiteson. QMIX: Monotonic value function factorisation for deep Multi-Agent reinforcement learning. March 2018.

[175] Sebastian Risi and Julian Togelius. Neuroevolution in games: State of the art and open challenges. *IEEE Transactions on Computational Intelligence and AI in Games*, 9(1):25–41, 2017.

[176] Glen Robertson and Ian Watson. A review of real-time strategy game ai. *AI Magazine*, 35(4):75–204, 2014.

[177] K Rogers and Andrew Skabar. A micromanagement task allocation system for real-time strategy games. *Computational Intelligence and AI in Games, IEEE Transactions on*, 2014.

[178] Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering Cambridge, England, 1994.

[179] Andrei A Rusu, Sergio Gomez Colmenarejo, Caglar Gulcehre, Guillaume Desjardins, James Kirkpatrick, Razvan Pascanu, Volodymyr Mnih, Koray Kavukcuoglu, and Raia Hadsell. Policy distillation. *arXiv preprint arXiv:1511.06295*, 2015.

[180] Franisek Sailer, Michael Buro, and Marc Lanctot. Adversarial planning through strategy simulation. In *Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on*, pages 80–87. IEEE, 2007.

[181] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

[182] Antonio A Sánchez-Ruiz. Predicting the outcome of small battles in starcraft. In *ICCBR (Workshops)*, pages 33–42, 2015.

[183] William Saunders, Girish Sastry, Andreas Stuhlmueller, and Owain Evans. Trial without error: Towards safe reinforcement learning via human intervention. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, pages 2067–2069. International Foundation for Autonomous Agents and Multiagent Systems, 2018.

[184] Frederik Schadd, Sander Bakkes, and Pieter Spronck. Opponent modeling in real-time strategy games. In *GAMEON*, pages 61–70, 2007.

[185] Jonathan Schaeffer. A gamut of games. *AI Magazine*, 22(3):29, 2001.

[186] Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *Science*, 317(5844):1518–1522, 2007.

[187] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[188] Douglas Schneider and Michael Buro. Starcraft unit motion: Analysis and search enhancements. In *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2015.

[189] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 815–823, 2015.

[190] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International Conference on Machine Learning*, pages 1889–1897, 2015.

[191] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[192] Claude E. Shannon. Programming a computer for playing chess. *Philosophical Magazine (First presented at the National IRE Convention, March 9, 1949, New York, U.S.A.)*, Ser.7, Vol. 41, No. 314, 1950.

[193] Amirhosein Shantia, Eric Begue, and Marco Wiering. Connectionist reinforcement learning for intelligent unit micro management in starcraft. In *Neural Networks (IJCNN), The 2011 International Joint Conference on*, pages 1794–1801. IEEE, 2011.

[194] Alexander Shleyfman, Antonín Komenda, and Carmel Domshlak. On combinatorial actions and cmabs with linear side information. In *ECAI*, pages 825–830, 2014.

[195] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[196] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[197] Stephen J Smith, Dana Nau, and Tom Throop. Computer Bridge: A big win for AI planning. *AI magazine*, 19(2):93, 1998.

[198] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*, 2014.

[199] Marius Stanescu. Rating systems with multiple factors. Master's thesis, School of Informatics, Univ. of Edinburgh, Edinburgh, UK, 2011.

[200] Marius Stanescu, Nicolas A. Barriga, and Michael Buro. Hierarchical adversarial search applied to real-time strategy games. In *Proceedings of the Tenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, pages 66–72, 2014.

[201] Marius Stanescu, Nicolas A. Barriga, and Michael Buro. Introducing hierarchical adversarial search, a scalable search procedure for real-time strategy games. In *Proceedings of the Twenty-first European Conference on Artificial Intelligence (ECAI)*, pages 1099–1100, 2014.

[202] Marius Stanescu, Nicolas A. Barriga, and Michael Buro. Using Lanchester attrition laws for combat prediction in StarCraft. In *Eleventh Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, pages 86–92, 2015.

[203] Marius Stanescu, Nicolas A. Barriga, and Michael Buro. Combat outcome prediction for real-time strategy games. In *Game AI Pro 3: Collected Wisdom of Game AI Professionals*, chapter 25. CRC Press, 2017.

[204] Marius Stanescu, Nicolas A. Barriga, Andy Hess, and Michael Buro. Evaluating real-time strategy game states using convolutional neural networks. In *IEEE Conference on Computational Intelligence and Games (CIG)*, 2016.

[205] Marius Stanescu and Michal Čertickỳ. Predicting opponent's production in real-time strategy games with answer set programming. *IEEE Transactions on Computational Intelligence and AI in Games*, 8(1):89–94, 2016.

[206] Marius Stanescu, Sergio Poo Hernandez, Graham Erickson, Russel Greiner, and Michael Buro. Predicting army combat outcomes in StarCraft. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2013.

[207] Statista. Number of players of selected eSports games worldwide as of august 2017. `https://www.statista.com/statistics/506923/esports-games-number-players-global/`, 2017 (accessed September 25, 2018).

[208] Statista. Worldwide eSports viewer numbers 2012-2021, by type. https://www.statista.com/statistics/490480/global-esports-audience-size-viewer-type/, 2017 (accessed September 25, 2018).

[209] Sainbayar Sukhbaatar, Arthur Szlam, and Rob Fergus. Learning multiagent communication with backpropagation. In D D Lee, M Sugiyama, U V Luxburg, I Guyon, and R Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 2244–2252. Curran Associates, Inc., 2016.

[210] Peng Sun, Xinghai Sun, Lei Han, Jiechao Xiong, Qing Wang, Bo Li, Yang Zheng, Ji Liu, Yongsheng Liu, Han Liu, et al. Tstarbots: Defeating the cheating level builtin ai in starcraft ii in the full game. *arXiv preprint arXiv:1809.07193*, 2018.

[211] Peter Sunehag, Guy Lever, Audrunas Gruslys, Wojciech Marian Czarnecki, Vinicius Zambaldi, Max Jaderberg, Marc Lanctot, Nicolas Sonnerat, Joel Z Leibo, Karl Tuyls, and Thore Graepel. Value-Decomposition networks for cooperative Multi-Agent learning. June 2017.

[212] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. The MIT Press, March 1998.

[213] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

[214] Omar Syed and Aamir Syed. Arimaa-a new game designed to be difficult for computers. *ICGA JOURNAL*, 26(2):138–139, 2003.

[215] Gabriel Synnaeve. *Bayesian programming and learning for multi-player video games*. PhD thesis, Université de Grenoble, 2012.

[216] Gabriel Synnaeve and Pierre Bessière. A Bayesian model for opening prediction in RTS games with application to StarCraft. In *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, pages 281–288. IEEE, 2011.

[217] Gabriel Synnaeve and Pierre Bessière. A Bayesian model for plan recognition in RTS games applied to StarCraft. *Proceedings of AIIDE*, pages 79–84, 2011.

[218] Gabriel Synnaeve and Pierre Bessière. A Bayesian model for plan recognition in RTS games applied to StarCraft. In AAAI, editor, *Proceedings of the Seventh Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2011)*, Proceedings of AIIDE, pages 79–84, Palo Alto, États-Unis, October 2011.

[219] Gabriel Synnaeve and Pierre Bessière. A dataset for StarCraft AI & an example of armies clustering. In *AIIDE Workshop on AI in Adversarial Real-time games 2012*, 2012.

[220] Gabriel Synnaeve and Pierre Bessière. Special tactics: a Bayesian approach to tactical decision-making. In *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, pages 409–416, 2012.

[221] Gabriel Synnaeve and Pierre Bessière. Multiscale Bayesian modeling for RTS games: An application to StarCraft AI. *IEEE Transactions on Computational intelligence and AI in Games*, 8(4):338–350, 2016.

[222] Christian Szegedy, Sergey Ioffe, and Vincent Vanhoucke. Inception-v4, inception-resnet and the impact of residual connections on learning. *arXiv preprint arXiv:1602.07261*, 2016.

[223] Ming Tan. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the tenth international conference on machine learning*, pages 330–337, 1993.

[224] Anderson Tavares, Hector Azpurua, Amanda Santos, and Luiz Chaimowicz. Rock, paper, starcraft: Strategy selection in real-time strategy games. In *The Twelfth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-16)*, 2016.

[225] Unity Technologies. Unity ml-agents toolkit (beta). `https://github.com/Unity-Technologies/ml-agents`, 2018 (accessed September 25, 2018).

[226] Chen Tessler, Shahar Givony, Tom Zahavy, Daniel J Mankowitz, and Shie Mannor. A deep hierarchical approach to lifelong learning in minecraft. In *AAAI*, volume 3, page 6, 2017.

[227] Michael Thielscher. Answer Set Programming for single-player games in general game playing. In *Logic Programming*, pages 327–341. Springer, 2009.

[228] Yuandong Tian, Qucheng Gong, Wenling Shang, Yuxin Wu, and C Lawrence Zitnick. Elf: An extensive, lightweight and flexible research platform for real-time strategy games. In *Advances in Neural Information Processing Systems*, pages 2659–2669, 2017.

[229] John Tromp. Solving connect-4 on medium board sizes. *ICGA JOURNAL*, 31(2):110–112, 2008.

[230] Alberto Uriarte. *Adversarial Search and Spatial Reasoning in Real Time Strategy Games*. PhD thesis, Drexel University, 2017.

[231] Alberto Uriarte and Santiago Ontañón. Kiting in RTS games using influence maps. In *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2012.

[232] Alberto Uriarte and Santiago Ontañón. Game-tree search over high-level game states in RTS games. In *Proceedings of the Tenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, AIIDE'14, pages 73–79, 2014.

[233] Alberto Uriarte and Santiago Ontañón. High-level representations for game-tree search in RTS games. In *Artificial Intelligence in Adversarial Real-Time Games Workshop, Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*, pages 14–18, 2014.

[234] Alberto Uriarte and Santiago Ontañón. Combat models for rts games. *IEEE Transactions on Computational Intelligence and AI in Games*, 2017.

[235] Nicolas Usunier, Gabriel Synnaeve, Zeming Lin, and Soumith Chintala. Episodic exploration for deep deterministic policies: An application to starcraft micromanagement tasks. In *5th International Conference on Learning Representations*, 2017.

[236] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Kuttler, John Agapiou, Julian Schrittwieser, Stephen Gaffney, Stig Petersen, Karen Simonyan, Tom Schaul, Hado van Hasselt, David Silver, Timothy Lillicrap, Kevin Calderone, Paul Keet, Anthony Brunasso, David Lawrence, Anders Ekermo, Jacob Repp, and Rodney Tsing. StarCraft II: A new challenge for reinforcement learning. Technical report, DeepMind, Blizzard, 2017.

[237] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. Dueling network architectures for deep reinforcement learning. 2015.

[238] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[239] Ben G. Weber and Michael Mateas. Case-based reasoning for build order in real-time strategy games. In *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment Conference, AAAI Press*, pages 106–111, 2009.

[240] Ben G. Weber and Michael Mateas. A data mining approach to strategy prediction. In *IEEE Symposium on Computational Intelligence and Games (CIG)*, 2009.

[241] Ben G. Weber, Michael Mateas, and Arnav Jhala. Applying goal-driven autonomy to StarCraft. In *Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 2010.

[242] Ben G. Weber, Michael Mateas, and Arnav Jhala. A particle model for state estimation in real-time strategy games. In *Proceedings of AIIDE*, page 103–108, Stanford, Palo Alto, California, 2011. AAAI Press, AAAI Press.

[243] Ben George Weber, Michael Mateas, and Arnav Jhala. Learning from demonstration for goal-driven autonomy. In *AAAI*, 2012.

[244] Jean-Christophe Weill. The ABDADA distributed minimax search algorithm. In *Proceedings of the 1996 ACM 24th annual conference on Computer science*, pages 131–138. ACM, 1996.

[245] Stefan Wender and Ian Watson. Applying reinforcement learning to small scale combat in the real-time strategy game StarCraft:Broodwar. In *CIG (IEEE)*, 2012.

[246] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[247] Steven Willmott, Julian Richardson, Alan Bundy, and John Levine. Applying adversarial planning techniques to Go. *Theoretical Computer Science*, 252(1):45–82, 2001.

[248] Andrew R. Wilson. Masters of war: History's greatest strategic thinkers. http://www.thegreatcourses.com/tgc/courses/course_detail.aspx?cid=9422, 2012.

[249] Samuel Wintermute, Joseph Z. Joseph Xu, and John E. Laird. SORTS: A human-level approach to real-time strategy AI. In *AI and Interactive Digital Entertainment Conference, AIIDE (AAAI)*, pages 55–60, 2007.

[250] Yuhuai Wu, Elman Mansimov, Shun Liao, Alec Radford, and John Schulman. OpenAI baselines: ACKTR & A2C. `https://blog.openai.com/baselines-acktr-a2c/`, 2017 (accessed September 25, 2018).

[251] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *CoRR*, abs/1505.00853, 2015.

[252] Yaodong Yang, Rui Luo, Minne Li, Ming Zhou, Weinan Zhang, and Jun Wang. Mean field multi-agent reinforcement learning. *arXiv preprint arXiv:1802.05438*, 2018.

[253] Lauren J. Young. What has IBM Watson been up to since winning 'Jeopardy!' 5 years ago? `https://www.inverse.com/article/13630-what-has-ibm-watson-been-up-to-since-winning-jeopardy-5-years-ago`, 2016 (accessed September 25, 2018).

[254] Vinicius Zambaldi, David Raposo, Adam Santoro, Victor Bapst, Yujia Li, Igor Babuschkin, Karl Tuyls, David Reichert, Timothy Lillicrap, Edward Lockhart, et al. Relational deep reinforcement learning. *arXiv preprint arXiv:1806.01830*, 2018.

[255] Xiang Zhang and Yann LeCun. Text understanding from scratch. *arXiv preprint arXiv:1502.01710*, 2015.

[256] Lianmin Zheng, Jiacheng Yang, Han Cai, Weinan Zhang, Jun Wang, and Yong Yu. Magent: A many-agent reinforcement learning platform for artificial collective intelligence. *arXiv preprint arXiv:1712.00600*, 2017.

Appendix A

Other Research

This appendix contains abstracts of further work I contributed to as part of my doctoral research. It is not included in the main body of this dissertation because my contributions were minor.

A.1 Parallel UCT Search on GPUs

This section contains the abstract of work led by Nicolas A. Barriga. My main contributions were in feedback relating to algorithm design and article writing. Michael Buro supervised the work.

We propose two parallel UCT search (Upper Confidence bounds applied to Trees) algorithms that take advantage of modern GPU hardware. Experiments using the game of Ataxx are conducted, and the algorithm's speed and playing strength is compared to sequential UCT running on the CPU and *Block Parallel* UCT that runs its simulations on a GPU. Empirical results show that our proposed *Multiblock Parallel* algorithm outperforms other approaches and can take advantage of the GPU hardware without the added complexity of searching multiple trees.

A.2 Building Placement Optimization in Real-Time Strategy Games

This section contains the abstract of work led by Nicolas A. Barriga. My main contributions were in assisting with experiments design and execution, and article writing. Michael Buro supervised the work.

It was previously published [9] at the Workshop on Artificial Intelligence in Adversarial Real-Time Games, part of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE), 2014.

In this paper we propose using a Genetic Algorithm to optimize the placement of buildings in Real-Time Strategy games. Candidate solutions are evaluated by running base assault simulations. We present experimental results in SparCraft – a StarCraft combat simulator – using battle setups extracted from human and bot StarCraft games. We show that our system is able to turn base assaults that are losses for the defenders into wins, as well as reduce the number of surviving attackers. Performance is heavily dependent on the quality of the prediction of the attacker army composition used for training, and its similarity to the army used for evaluation. These results apply to both human and bot games.

Appendix B

Research Environments

Here we describe some of the most common RTS game AI research environments in current use.

B.1 StarCraft: Brood War

StarCraft is a military science fiction RTS game developed by Blizzard Entertainment and published in 1998. StarCraft: Brood War is the expansion pack published later in the same year. The game features three distinct races that require different strategies and tactics to succeed. The Protoss have access to powerful units with advanced technology and psionic abilities. However, these units are costly and slow to acquire, encouraging players to focus on strategies that rely more on unit quality than quantity. The Zerg, in contrast, have entirely biological units and building which are weaker, but faster and cheaper, forcing the player to rely on large unit groups to defeat the enemy. The Terran provide a middle ground between the other two races, offering more versatile units. The game revolves around players collecting resources to construct a base, upgrade their military forces, and ultimately destroy all opponents' structures.

The BWAPI library enables AI systems to play the game and compete against each other. This, coupled with the fact that there are professional players and thousands of replays available for analysis, have contributed to make StarCraft the leading platform for RTS game AI research.

Figure B.1: Screenshot of StarCraft:Brood War, showing a group of Zerg assaulting a Protoss base.

Some characteristics of StarCraft are:

Real-time: The game runs at 24 simulation frames per second, and it moves on even if a player does not execute any actions.

Simultaneous moves: All players can issue actions at every frame.

Durative actions: Some actions take several frames to complete.

Imperfect information: *Fog-of-War* prevents each player from seeing the terrain and enemy units until a unit under his command has scouted it. Moreover, only terrain remains visible, with all enemy activity hidden, in previously revealed areas without a friendly unit currently in the vicinity.

Map size: Common map sizes range between 64x64 and 256x256 build tiles. A build tiles is the basic map subdivision, used for placing building. Each build tile can be divided into 4x4 walk tiles, for determining walkable areas. Each walk tile is comprised of 8x8 pixels, which determine the precise location moving units.

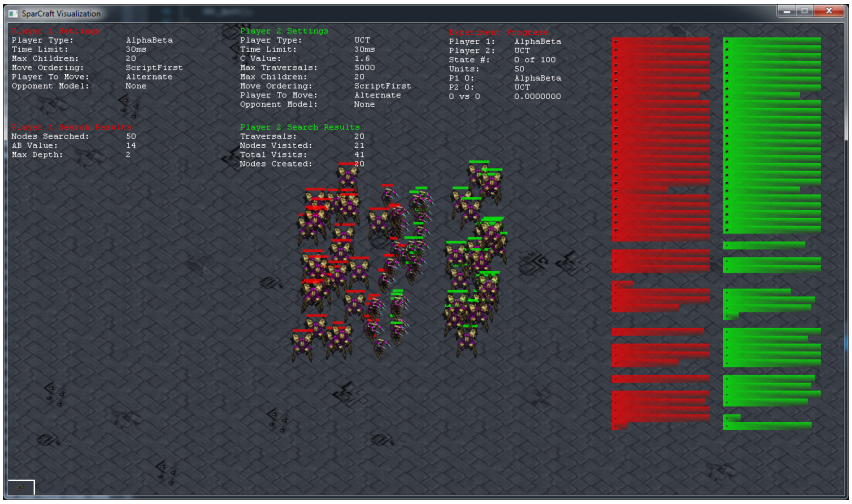


Figure B.2: Screenshot of SparCraft, showing a combat experiment between Alpha-Beta and UCT based players.

State/Action Space: [166] provide 10^{1685} as a lower bound on the number of possible states in StarCraft. They also estimate the average branching factor to be $\geq 10^{50}$.

B.2 SparCraft

SparCraft is a StarCraft combat simulator written by David Churchill[1]. Its main design goal was to estimate the outcomes of battles as quickly as possible. To accomplish this it greatly simplifies the game. There are no collisions, thus, pathfinding is not required. Only basic combat units are supported, and no spells, cloaking or burrowing. It comes with several scripted players implementing different policies, as well as several search based players. Adding missing features is possible, as SparCraft is an open source project.

B.3 μRTS

μRTS[2] is a simple RTS game designed to test AI techniques. It provides the basic features of RTS games, while keeping things as simple as possible: only four unit and two building types are supported, all of them with size

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

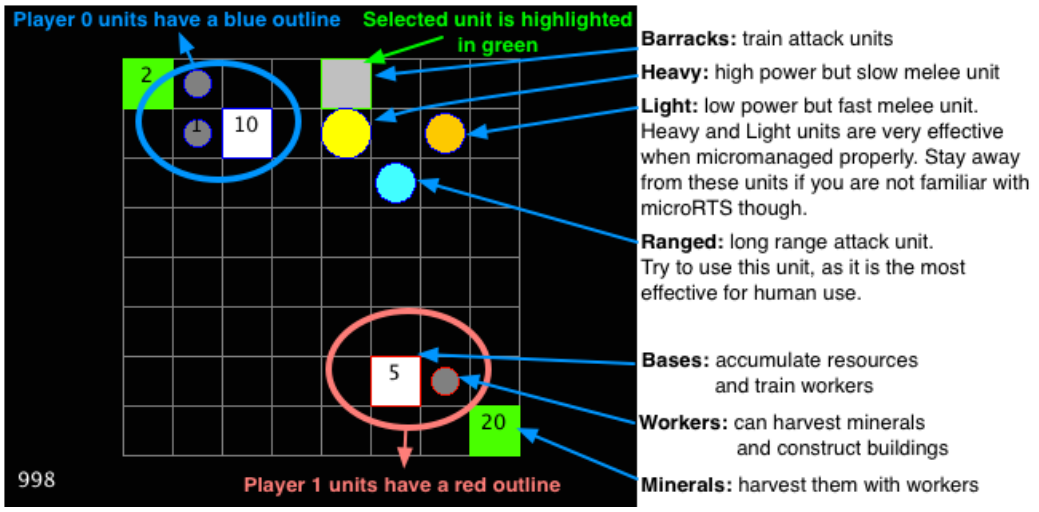


Figure B.3: Screenshot of μRTS, with explanations of the different in-game symbols.

of one tile. There is only one resource type. μRTS comes with a few basic scripted players, as well as search based players implementing several state-of-the-art RTS search techniques, making it a useful tool for benchmarking new algorithms.

Some of its characteristics are:

Real-time: With simultaneous and durative moves.

Perfect information: The game state is fully observable.

Map sizes: Configurable. Usual map sizes used in published papers range from 8x8 to 128x128.

B.4 MAgent

MAgent[3] is a research platform for many-agent reinforcement learning which allows for very large scale gridworld combat scenarios [256]. While other platforms suitable for use with reinforcement learning approaches can support only a few or even single agents, MAgent can scale up to a million of agents on the

[3] `https://github.com/geek-ai/MAgent`

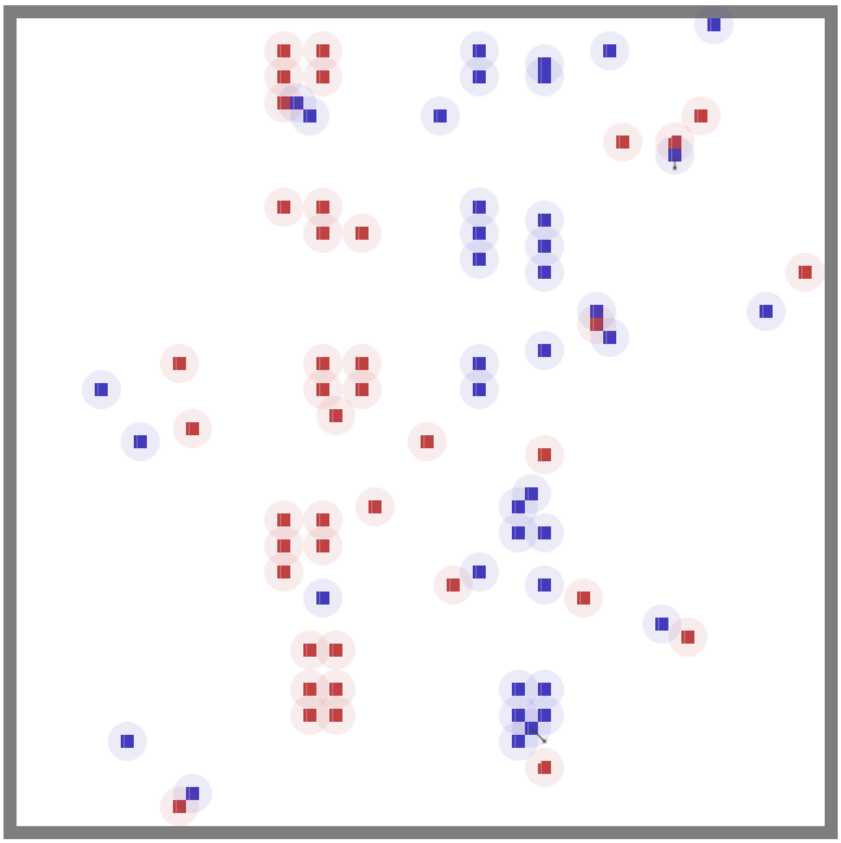


Figure B.4: MAgent scenario on a 64×64 map with 40 units on each side.

same map and still be computationally manageable. Moreover, it provides environment/agent configurations as well as a reward description language that enables flexible environment design.

Some of its characteristics are:

Real-time: With simultaneous moves.

Perfect information: Agents' view range is configurable.

Map sizes: Configurable. Obstacles and agents can be added in any pattern.

Others: Agent types and abilities are easily described via configuration files. Reward rules can be defined by constant attributes depending on agent type or by using triggers for game events. The framework provides observations in format directly interpretable as NN input, including both spatial and non spatial features.