Search, Abstractions and Learning in Real-Time Strategy Games

by

Nicolas Arturo Barriga Richards

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

Doctor of Philosophy

Department of Computing Science

University of Alberta

Abstract

Real-time strategy (RTS) games are war simulation video games in which the players perform several simultaneous tasks like gathering and spending resources, building a base, and controlling units in combat against an enemy force. RTS games have recently drawn the interest of the game AI research community, due to its interesting sub-problems and the availability of professional human players.

Large state and action space make standard adversarial search techniques impractical. Sampling the action space can lead to strong tactical performance on smaller scenarios, but doesn't scale to the sizes used on commercial RTS games. Using state and/or action abstractions contributes to solid strategic decision making, but tactical performance suffers, due to the necessary simplifications introduced by the abstractions. Combining both techniques is not straightforward, due to the real-time constraints involved.

We first present *Puppet Search*, a search framework that employs scripts as action abstractions. It produces agents with strong strategic awareness, as well as adequate tactical performance. The tactical performance comes from incorporating sub-problem solutions, such as pathfinding and build order search, into the scripts. We then split the available computation time between this strategic search and *NaïveMCTS*, a strong tactical search algorithm that samples the low-level action space. This second search refines the output of the first one by reassigning actions to units engaged in combat with the opponent's units. Finally, we present a deep convolutional neural network (CNN) that can accurately predict *Puppet Search* output in a fraction of the time, thus leaving more time available for tactical search.

Experiments in *StarCraft: Brood War* show that *Puppet Search* outperforms its component scripts, while in μRTS it also surpasses other state-of-the-art agents. Further experimental results show that the combined *Puppet Search*/NaïveMCTS algorithm achieves higher win-rates than either of its two independent components and other state-of-the-art μRTS agents. Finally, replacing *Puppet Search* with a CNN shows even higher performance.

To the best of our knowledge, this is the first successful application of a convolutional network to play a full RTS game on standard sized game maps, as previous work has focused on sub-problems, such as combat, or on very small maps. We propose further work to focus on partial observability and CNNs for tactical decision-making. Finally, we explore possible utilization in other game genres and potential applications on the game development process itself, such as playtesting and game balancing.

Preface

This article-based dissertation contains previously published material. The author of this dissertation is the first author of these papers, unless otherwise stated.

Chapter 3 is joint work with Marius Stanescu and Michael Buro. It was previously published [4] at the IEEE Conference on Computational Intelligence and Games (CIG), 2014.

Chapter 5 is joint work with Marius Stanescu and Michael Buro. It was previously published [3] 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.

Chapter 6 is joint work with Marius Stanescu and Michael Buro. It was previously published [5] at the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE), 2015.

Chapter 7 is joint work with Marius Stanescu and Michael Buro. It was previously published [8] in the IEEE Transactions on Computational Intelligence and AI in Games Journal, 2017.

Chapter 8 is joint work with Marius Stanescu and Michael Buro. It is accepted for presentation [7] at the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE), 2017.

In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of University of Alberta's products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/

To Pamela

For embarking on this adventure with me.

To Diego

For being the reason to improve myself.

Acknowledgements

I would first like to thank Professor Michael Buro for his invaluable guidance during my doctoral studies.

I would also like to thank my committee members Vadim Bulitko, Martin Müller, Nilanjan Ray and Santiago Ontañón for their feedback.

Special thanks to Marius Stanescu for the long hours toiling away together at more deadlines than I want to remember.

And last but not least, my profound gratitude to my family, without whom none of this would matter.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Even before the dawn of electronic computers, people have been trying to build machines able to play games. In 1769, Baron Wolfgang von Kempelen built a machine, called *The Turk*, capable of playing Chess at a very high level. After several decades of impressing people all around Europe and America, it was revealed to be a hoax: a human was operating the machine from a secret compartment all along, with the position filled by different chess masters throughout the years.

Even though *The Turk* was nothing more than an elaborate hoax, it shows the interest that automated game playing elicits among scientists and engineers, so much so, that only a few years after the first electronic computer was built, the first paper proposing a program to play a full game of chess was published: Claude Shannon's 1949 paper "Programming a Computer for Playing Chess" [79]. In it, Shannon motivates game AI research by stating 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 follows by citing possible related problems, ranging from electronic design to symbolic mathematical operations to military decision making, among others. His proposed solution was to use a minimax algorithm to look into possible future game states, and an approximate evaluation function to decide who is winning or losing in a certain state. This is a combination that is still used in most two-player, deterministic, perfect information game playing programs to date. The following year, Alan Turing wrote the first program to play chess, but didn't have any hardware capable of running it. A series of programs were developed subsequently that could play certain aspects of chess, until finally in 1957, Alex Bernstein created the first complete chess playing program.

Things have come a long way since then, with modern game playing programs able to defeat top Go professionals and Chess grandmasters. As AI techniques improve, game playing programs are defeating top human players in an increasing number of games. Some of those have even been solved, like Checkers [78] and Heads-up Limit Hold'em Poker [11]. Others still, have proven more resilient, like Skat and Bridge, with top game playing programs approaching, but not yet reaching, professional level strength.

Besides Shannon's motivations of applicability to other domains, computer board games are a small but non negligible niche market. However, the modern video game industry presents a big commercial incentive for AI research. The global video game market is close to $100 billion, with the US and China surpassing $25 billion each[1]. Video game AI applications range from tools to help game design (testing, balancing, etc), to algorithms that enable new forms of game play (such as pathfinding), to building non-player characters (NPCs) to be used either as an opponent or in collaboration with human players.

Video games introduce a host of new challenges not usually present in board games: actions can be issued at any time by either player (simultaneous moves), at a rate of several actions per second (real-time), those actions are not instantaneous, they take some time to complete (durative actions), in some games the result of those actions is stochastic, most video games present only a partial view of the game state to the players (imperfect information) and the size of the playing area, the number of units available and the possible actions at any given time are several orders of magnitude bigger than most board games.

Besides these challenges, video games offer us an opportunity that very few recent board games offer: professional human players. The rise of professional *eSports* communities enables us to seriously engage in the development of competitive AI for video games. A game played just by amateurs could look interestingly difficult at first glance, but top players might be easily defeated with just a little research. An example of this is Arimaa [91], which was purposely designed to be difficult for computers but easy for human players. It took a decade for game playing programs to defeat the top human players [90], but it needed no new techniques other than those already in use in Chess and Go programs. And after those ten years, we don't know if the computer players are really strong, or it's just that no human player has a truly deep understanding of the game. A game that has a large professional player base provides a more solid challenge.

One particularly interesting game category or genre within video games is Real-Time Strategy (RTS) games. These games present some similarities to traditional abstract strategy games, such as Chess or Go. They are zero-sum games, played in a grid-world, and have units with strict rules on what kind of actions they can perform, so there is hope that some of the existing game AI techniques will at least serve as a starting point to build agents capable of playing these games well. RTS games also present most of the characteristics mentioned in the previous paragraphs, as well as others, such as partial game state visibility, resource management, possibly multiple opponents or allies, and a vastly bigger state-space and game-tree complexity [69].

RTS games provide us with many research problems with considerable state spaces and large numbers of actions, in which decisions have to be made under

[1] `https://newzoo.com/resources/`

real-time constraints. These problems force us to think about abstractions if we want to make standard search algorithms work, potentially having a big impact on other real-time decision problems like autonomous driving, emergency services dispatching or assisting air, rail or subway traffic controllers.

Despite the similarities between RTS games and abstract strategy games, there is a big gap in state-space and game-tree complexity that has to be overcome if we are to successfully apply traditional AI techniques. Existing RTS game playing agents rely on a combination of search, machine learning and expert knowledge for specific sub-problems, but no unified search approach has yet been implemented to play a full scale RTS game. Such an approach would rely on either machine learning or state forwarding for predicting outcomes, rather than on ad-hoc heuristics for deciding which actions to take. I propose a search framework for RTS games that uses high-level abstract actions representing strategies. Using abstractions would reduce the search space enough to make search tractable, and would integrate previous sub-problem solutions by treating them either as black-boxes or parameterized scripts. If the action abstractions prove to be too coarse to handle tactical issues, a second search or ML based algorithm can be used to refine actions in critical spots.

1.1 Contributions

The main contributions in this dissertation are:

- Specific RTS sub-problem solutions that can be included in hand-coded configurable scripts, to be used as an action abstraction mechanism.

- An adversarial look-ahead search framework that uses abstract actions to build strong RTS game playing agents.

- Deep convolutional neural networks, as a fast alternative to adversarial look-ahead search, for choosing high-level actions in RTS game playing agents.

- Combined strategic and tactical algorithms that produce stronger RTS game playing agents than either on its own.

We expect agents based on search or learning over configurable scripts to perform well in any videogame for which scripted AI systems can be built, like first-person shooters (FPS) or multiplayer online battle arenas (MOBAs).

We trust our script-based techniques to be valued by commercial game AI designers. It allows them to retain control over the range of behaviours the AI system can perform, while at the same time, the search (or prediction) component enables the system to decide on actions based on predicted outcomes. This dual advantage leads to a stronger, and hopefully more believable and enjoyable opponent.

Game playing agents can also be used for automated playtesting and game balancing. The most tedious parts of playtesting often involve trying and repeating slight variations of game play, looking for bugs in edge cases. Search algorithms are good at this, given that the way they work is by exploring the game space. A further application for AI agents is game balancing. Every time game parameters — such as the health or cost of a particular unit — are changed, the game needs to undergo a significant amount of testing, to ensure that the change produces the expected results, and doesn't have any unexpected consequences. This testing can be automated by having AI agents play the game. Similar agents could also be used to assist PCG and AI-assisted design, to evaluate both human and AI generated content.

Possible applications extend to areas besides videogames. Automated tools are increasingly being used in several phases of the military decision making process [72]. One such phase is *Course of Action* (COA) analysis, a time-consuming process by which highly specialized brigade or division staff plan and schedule tasks, allocate resources, estimate attrition and predict enemy actions. This adversarial and time-sensitive planning includes high-level durative actions, as well as scripted assumptions about enemy behaviour; in short, it is a perfect domain for applying adversarial search techniques over action abstraction scripts. In fact, similar systems already exist [53], although with limited capabilities. Furthermore, these systems already make use of high-level simulations to predict outcomes [43], as we will present in chapter 6.

There is nothing in the idea of using configurable scripts as an action abstraction mechanism that is particular to adversarial environments. The basic requisite is that multiple non-optimal heuristic solutions can be created. Also needed is a reasonably fast forward model for search, or a learning procedure to decide on the best action. Example applications might include classic single agent planning problems such as the travelling salesman or vehicle routing problems. A more unconventional application might be a real-world implementation of a blocks world problem, where a robot needs to move a set of blocks from a starting configuration to a goal, while minimizing some resource usage, such as time or distance travelled. A few different greedy heuristics can be constructed to find sub-optimal solutions. These heuristics solutions — or scripts — will abstract from the mechanical details of the robot, such as how to go in a straight line from point A to B, or how to lift a cube. They can expose high-level choices to a search or learning procedure, which will then decide the best action to take among the scripts and choices available.

1.2 Outline

Chapter 2 presents an overview of previous research on RTS game AI, outlining several sub-problems for which existing solutions can be found in the literature. Chapter 3 presents an MCTS variant that takes advantage of GPU hardware. Chapter 4 summarizes a hierarchical adversarial search framework for RTS games. Chapter 5 presents a genetic algorithm for placing building in bases. Chapter 6 introduces *Puppet Search*, a game tree search algorithm over abstract actions, represented by scripts with choices. Chapter 7 extends the *Puppet Search* analysis and compares it to other state-of-the-art adversarial search algorithms. Chapter 8 presents value and policy deep convolutional neural networks (CNN) for evaluating RTS game states and strategy selection respectively. It also shows an RTS agent using the policy network for strategic decisions and a search algorithm with the value network for tactical decisions. It showcases the research undertaken by the author both towards solving subproblems, and towards a unifying search framework. Chapter 9 synthesizes and combines the conclusions of the previous chapters, and discusses ideas for future research and applications.

Abstracts of further research in which I was involved can be found in Appendix A.

Chapter 2

Literature Survey

During the last decade StarCraft: Brood War has emerged as the *de facto* testbed in Real-Time Strategy games research. The appeal stems in part from the availability of BWAPI[1], a programming API that lets AI systems play the game, and in part from its huge player base, including professional leagues[2]. StarCraft is a complex RTS game, featuring two resource types, ground and air units, melee and ranged, and three different races, with apparently no dominant race or strategy. For a detailed description of StarCraft see appendix B.1. Several Brood War AI competitions are held every year, most notable among them being the ones attached to the AIIDE[3] and CIG[4] conferences, as well as the Student StarCraft AI Tournament[5] (SSCAI).

A few other research platforms have recently arisen, providing more restricted scenarios: 1. μ**RTS** is a small academic RTS game, deterministic, fully-observable, with simultaneous and durative actions, described in detail in appendix B.3; 2. **SparCraft** is a StarCraft combat simulator, with simplifications to increase simulation speed, described in appendix B.2.

Some research games used in the past have fallen out of favour since the release of BWAPI in 2009: 1. **Wargus**, an open source WarCraft 2 clone; 2. **SpringRTS**, a 3D RTS game engine; 3. **ORTS**, an academic RTS engine.

In late 2016, Blizzard announced that it was working with DeepMind on an API for StarCraft II. This has created significant anticipation in the research community, and it is widely expected that this will become a major RTS AI research platform. This API was released publicly on August 9th 2017, along with a reinforcement learning framework [107].

[1] https://github.com/bwapi/bwapi

[2] http://wiki.teamliquid.net/starcraft/Leagues

[3] http://www.starcraftaicompetition.com/

[4] http://cilab.sejong.ac.kr/sc_competition/

[5] http://sscaitournament.com/

2.1 RTS Game Sub-Problems

Building an RTS game playing agent can be decomposed into several sub-problems, ranging from finding a high-level strategy to win a game to individual unit movement. Different researchers classify these sub-problems in slightly different categories.

Robertson and Watson [74] propose two broad categories:

- **Strategic Decision Making** or *macromanagement*, dealing with long-term high-level planning and decision-making. It includes resource allocation, production scheduling and major military decisions.
- **Tactical Decision Making** or *micromanagement*, encompasses short term individual unit control and medium-term planning with groups of units.

Ontañón *et al.* [69] instead offer three categories:

- **Strategy**, which concerns devising and/or choosing strategies, opponent modelling, and build order planning.
- **Tactics**, covering terrain analysis, building placement, scouting and higher level aspects of combat.
- **Reactive Control**, involving pathfinding, navigation and individual unit movement and combat.

In a later paper [68], Ontañón *et al.* suggests that decomposition into sub-problems is only one of six challenge areas in RTS AI research:

- **Planning**, includes both long-term plans for economic and strategic development, and short-term plans for low level unit control.
- **Learning**, divided in three types: **Prior learning:** exploiting available data, such as replays; **In-game learning:** improving while playing a game, using techniques such as reinforcement learning or opponent modelling; **Inter-game learning:** using experience from one game, to increase the chance of victory in the next one.
- **Uncertainty**, tackling the partial observability of the game, as well as the adversarial nature of it.
- **Spatial and Temporal Reasoning**, terrain exploitation (building placement, base expansions, unit positioning) for tactical reasoning, and timing builds and attacks for strategic reasoning.
- **Domain Knowledge Exploitation**, exploiting available knowledge, e.g. by data-mining replays or hard-coding strategies from guides and forums.

- **Task Decomposition**, dividing the game into sub-problems, as was done in the previous paragraphs.

2.2 High-Level Approaches

Very little focus has been given to unifying the solutions to the different sub-problems. A few high-level approaches have been tried, but they have been of limited scope. Stanescu *et al.* [83, 84] propose Hierarchical Adversarial Search in which each search layer works at a different abstraction level, and each is given a goal and an abstract view of the game state. For search purposes the game is advanced at the lowest level and the resulting states are abstracted back up the hierarchy. This algorithm was tested in SparCraft, and though this algorithm is general enough to encompass a full RTS game, only combat-related experiments were conducted. More details about this algorithm are presented in Chapter 4.

Another proposal, by Uriarte and Ontañón [103, 104], followed a different path, by abstracting the game state, searching at the highest abstraction level, and then translating the results back to the lower level for execution. The state representation uses a decomposition of the map into connected regions and groups all units of the same type into squads in each region. Moves are restricted to squads, which can move to a neighbouring region, attack, or stay idle. This approach only deals with combat, however, and was added as a module to an existing StarCraft bot. Results were only presented for playing against the built-in AI which is much weaker than state-of-the-art bots.

Finally, Ontañón and Buro [65] present Adversarial Hierarchical Task Network (AHTN) planning, which combines game tree search with HTN planning. 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.

2.3 Strategic Planning

A strategy, or strategic plan, is a high-level plan of action or policy executed by an agent to accomplish a given goal. Strategic planning involves both recognizing the opponent's plan and producing one of our own.

2.3.1 Strategy Selection

Strategy selection has been tackled using a wide variety of techniques, from scripted expert knowledge to machine learning.

The most popular approach in commercial video games is to use Finite State Machines (FSM) [46] which let the authors specify the desired behaviour. This approach is simple to implement and gives total control of the agent's behaviour to the designer. The main drawback is the lack of adaptation; once an opponent has found a weakness in the agent, it can exploit it over and over again.

Ontañón *et al.* [67] and Mishra *et al.* [59] use Case-Based Planning (CBP) to retrieve and adapt plans learned by observing human replays in the game of Wargus. This approach still has the problem that it cannot learn from its mistakes. If a weakness is found, it can be exploited by an opponent until a new case is added to its case library showing it how to act in that situation. This is an advantage over the FSM approach, because it is usually easier to show a behaviour in the game than to code it.

Elogeel *et al.* [33] try to overcome the exploitability issue by Concurrent Plan Augmentation (CPA), which adds contingency plan branches where the winning probability is low according to a simulator. The contingency branches will be used in case the situation diverges from what the original plan expected.

Chung *et al.* [18] and Sailer *et al.* [76] propose systems to choose among a set of predefined strategies. The former performs Monte-Carlo simulations, using randomly chosen strategies for both players, and picking the one with the best overall statistical results. The latter uses simulations to fills out a pay-off matrix for both players' strategies and chooses the Nash-optimal one.

2.3.2 Opponent Modelling

Opponent modelling can be used to aid in the strategy selection process, or to exploit opponents' weaknesses. Several machine learning techniques for this have been proposed over the last decade.

Schadd *et al.* [77] propose an approach to opponent modelling based on hierarchically structured models. The top-level of the hierarchy can classify the general play style of the opponent while the bottom-level of the hierarchy classifies specific strategies that further define the opponent's behaviour. Hsieh and Sun [47] used CBR to predict human strategies using a case library built from human game replays. Weber and Mateas [110] applied several machine learning techniques to predict the opponent's strategy and to forecast when specific units or buildings will be produced.

Dereszynski *et al.* [31] used Hidden Markov Models (HMM) to learn build order probabilities, and construct an opponent model useful for predicting opponent behaviour or identifying unusual or novel strategies. Synnaeve and Bessière [93] use a Bayesian model to learn from StarCraft replays and predict opening strategies. The same authors also [94] use a similar system for predicting the technology level of the opponent.

Certicky and Certicky [15] used CBR for selecting the most effective army composition based on the available information about the enemy's army composition.

Stanescu and Certicky [88] employ answer set programming (ASP) to predict the most probably combination of units produced by the opponent in a given time span. They report a high accuracy at predicting production for 1–3 minute intervals in the games of StarCraft and WarCraft III.

2.3.3 Build Order Planning

Build order planning in RTS games is the problem of finding action sequences that create a certain number of buildings and units. The solutions are constrained by prerequisites and resources, and can be optimized on different criteria, such as total build time or resources needed. In RTS game playing, two main sub-problems can be identified relating to build order planning: selecting a build order that maximizes the winning chances against an opponent, and optimizing some aspect of the build order, like cost or time to complete.

Weber and Mateas [109] use case-based reasoning for selecting build orders from a case library. They performed their experiments in the game of Wargus, using perfect and imperfect information setups. The case library was built by playing eight different scripts against each other and saving the cases for the winning scripts. When playing the CBR based agent against four other scripts, they show a big performance improvement over just playing one random script out of the eight training ones. Furthermore, the gains were more noticeable in the imperfect information setup.

Churchill and Buro [22] tackle a slightly different problem: rather than trying to find a build order to defeat the enemy, their algorithm takes a set of units and buildings to be constructed (a goal) as input, and returns a sequence of units and buildings that need to be produced to meet prerequisites and gather resources to reach the goal. Depth-first branch and bound is used to optimize the plan duration, the time it take to build all the units and buildings in the plan. They provide experimental results showing plan durations comparable to those of professional StarCraft players, while keeping the computational cost low.

2.3.4 Battle Planning

Combat in RTS games is usually defined as a tactical problem, but the decision of whether, when and where to attack has important strategic consequences. When deciding to attack the opponent, an attack location has to be selected. This decision has to take into account both armies and bases spacial distributions as well as their compositions. For example, one could face the opponent's army directly, to avoid having one's base assaulted while attacking. Or one could go and try to attack an undefended enemy base, hoping to destroy it before their army can arrive to rescue it. To my knowledge, no research has been published on selecting attack locations or coordinating multiple combat scenarios.

2.4 Tactical Planning

Tactical planning involves decisions with more localized effects than strategic planning. Their direct area of influence is usually limited in space and time, though the consequences can affect the full game.

2.4.1 Building Placement

A sub-problem of RTS game AI that has seen little research is building placement, which is concerned with constructing buildings at strategic locations with the goal of slowing down potential enemy attacks as much as possible while still allowing friendly units to move around freely. Expert human players optimize their base layouts, whereas current programs do not and therefore are susceptible to devastating base attacks. Finding good building locations is difficult. It involves both spatial and temporal reasoning, and ranges from blocking melee units completely [16, 73] to creating bottlenecks or even maze-like configurations that maximize the time invading units are exposed to own static and mobile defenses.

Important factors for building placement are terrain features (such as ramps and the distance to expansion locations), the cost of constructing static defenses, and the type of attacking enemy units. Expert human players are able to optimize building locations by applying general principles such as creating choke-points, and then refining placement in the course of playing the same maps over and over and analyzing how to counter experienced opponent attacks. Methods used in state-of-the-art RTS bots are far less sophisticated [69]. Some programs utilize the terrain analysis library BWTA [71] to identify chokepoints and regions to decide where to place defenses. Others simply execute pre-defined building placement scripts authors have devised for specific maps. An even simpler approach is to use a simple spiraling search to find legal building locations in a particular base.

Barriga *et al.* [3] use a Genetic Algorithm with combat simulations as fitness functions to optimize building locations. More details about this approach can be found in Chapter 5.

2.4.2 Combat and Unit Micromanagement

Unit micromanagement is the selection of specific actions for units to perform. It is especially important during combat and can often be the deciding factor in a game. Combat in RTS games is a hard problem, Furtak and Buro [37] have shown that even the sub-problem of computing the existence of deterministic winning strategies in a basic shooting game is PSPACE-hard and in EXPTIME.

Search Algorithms

The first attempt at using search for RTS combat was the *randomized alpha-beta search* (RAB) algorithm [54]. It was an adaptation of alpha-beta search for games with simultaneous moves, with experiments in an abstract shooting game in which all units are static and can shoot any enemy unit at any given time. Churchill and Buro [27] propose *Alpha-Beta Considering Durations* (ABCD), an alpha-beta adaptation for games with simultaneous and durative moves, and showed good performance in small combat scenarios (between 2 and 8 units per side) using SparCraft. Lastly, Portfolio Greedy Search [25] reduces the action space by only considering moves suggested by a set of scripts (the portfolio), rather than all possible actions by every unit. Experiments performed in SparCraft against ABCD and UCTCD (a UCT adaptation similar to ABCD) shows that it vastly outperforms them for large combat scenarios (between 16 and 50 units per side).

Ontañón [62, 64] treats the problem of assigning actions to individual units as a Combinatorial Multi-Armed Bandit (CMAB) problem, that is, a bandit problem with multiple variables. Each variable represents a unit, and the legal actions for each of those units are the values that each variable can take. He proposes to treat each variable separately, rather than translating it to a regular Multi-Armed Bandit (MAB) (by considering that each possible legal value combination is a different arm) as in UCTCD. His strategy is to sample the CMAB, assuming that the reward of the combination of the arms is just the sum of the rewards for each arm (which they call the *naïve assumption*). He treats each arm as an independent local MAB for exploration purposes, and combines the individual samples for each arm in a global MAB during exploitation. The algorithm (naïveMCTS) was compared against other algorithms that sample or explore all possible low-level moves, such as ABCD and UCTCD. It outperformed them all, in three μRTS scenarios, with the biggest advantages found on the more complex scenarios. The author then adds a Bayesian model to calculate prior probabilities in InformedMCTS [63]. Shleyfman *et al.* [80] propose a new sampling method that doesn't rely on the *naïve assumption*, by exploiting the fact that the unit action's rewards are not independent. They show μRTS experiments outperforming naïveMCTS.

Balla and Fern [2] applied UCT to tactical assault planning. They worked with groups of units, rather than individual units, and just two actions, *join* with a friendly group, or *attack* an enemy group. Uriarte and Ontañón [102] used influence maps to implement kiting (a common technique in RTS games to shoot an enemy unit while trying to maintain a safe distance to it), showing good results in a full StarCraft bot.

Machine Learning

Wender and Watson [113] used reinforcement learning for micromanaging combat units in StarCraft. The authors showed that an AI agent using RL is able to learn a strategy that performs well in small combat scenarios, though they only tested it against the rather weak built-in AI. The main drawback of the approach is that the actions had to be simplified to just two: Fight and Retreat.

Stanescu *et al.* [85] present a method for estimating combat outcomes in StarCraft, and apply it in a bot to decide whether to engage the enemy or retreat. The method is an extension to Lanchester's laws of combat [56], to calculate the attrition rates of opposing armies. Lanchester's original laws assume each army is composed of identical soldiers. As in RTS battles this is rarely the case, support for different unit types was added, as well as accounting for injured units. This system is faster than running simulations in SparCraft, and leads to better results against state-of-the-art bots.

Deep Learning

Deep learning [42] was popularized a few years ago by revolutionary results [55] in the ImageNet competition. It promptly found its way into videogame AI agents, where a deep convolutional neural network trained by Q-learning achieved humal level performance in Atari games [61].

Last year, Usunier *et al.* [106] presented a deep network that outperforms a set of simple baseline heuristics in StarCraft micromanagement tasks. The network managed to learn simple policies, like focusing fire, from self play. Sukhbaatar *et al.* [89] propose CommNet, a model in which each agent is controlled by a network with continuous communication, where the communication is learned alongside the policy. This model outperforms models without communication, fully connected ones, or models with discrete communication, in combat tasks. BiCNet [70] adds support for heterogeneous agents using bidirectional communication and recurrent neural networks.

2.5 Further Reading

In addition to the three major surveys cited earlier [69, 74, 68], there have been a few doctoral dissertations published with a strong focus on RTS games. The most notable among them on heuristic search techniques [21] and adversarial search and spatial reasoning [101].

Chapter 3

Parallel UCT Search on GPUs

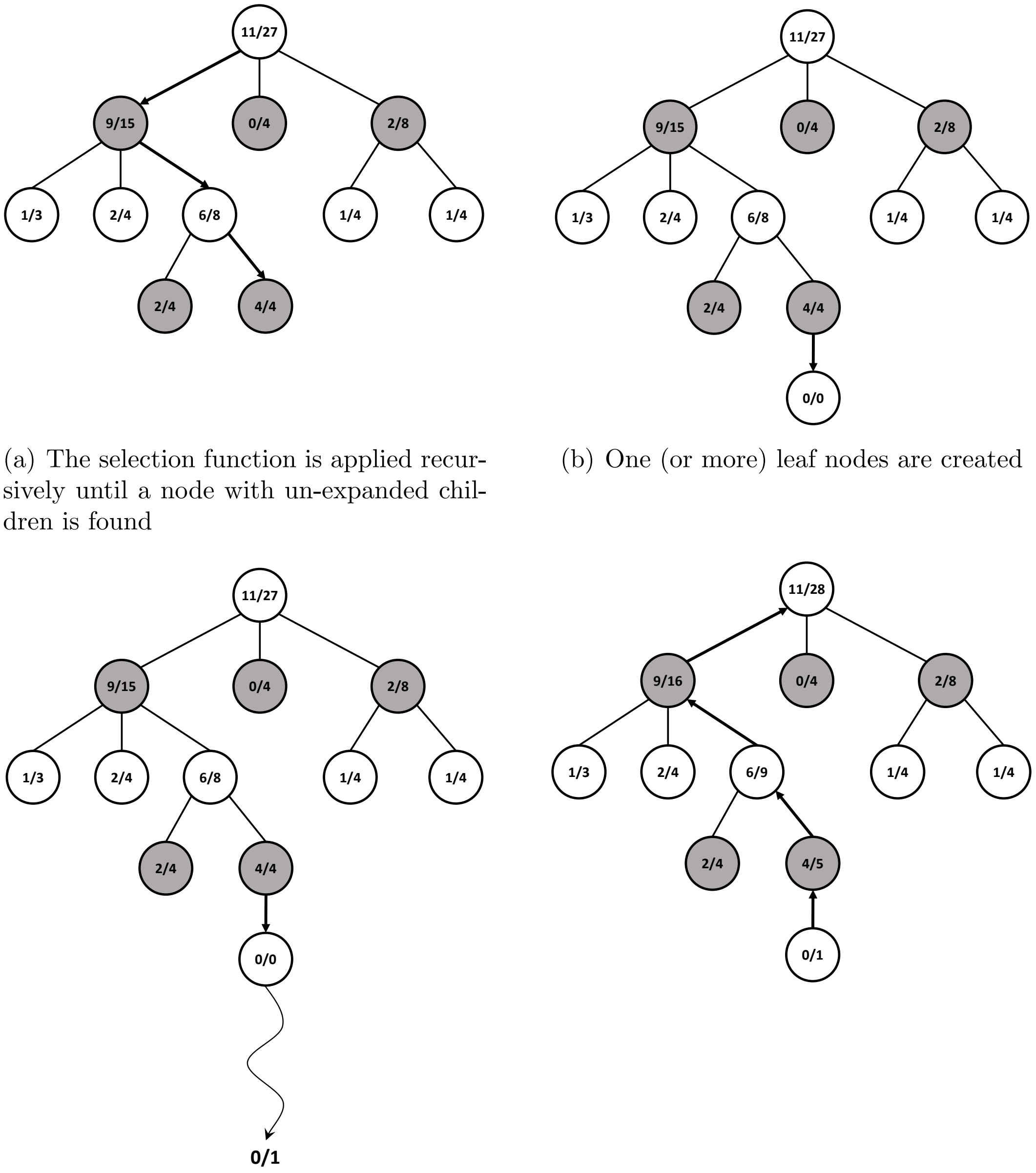
Abstract

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.

3.1 Introduction

Monte-Carlo Tree Search (MCTS) has been very successful in two player games for which it is difficult to create a good heuristic evaluation function. It has allowed Go programs to reach master level for the first time (see [29] and [38]). Recent results include a 4-stone handicap win by Crazy Stone against a professional player [112].

MCTS consists of several phases, as shown in Figure 3.1, which for our purposes can be classified as in-tree and off-tree. During the in-tree phases, the algorithm needs to select a node, expand it, and later update it and its ancestors. The off-tree phase consists of possibly randomized playouts starting from the selected node, playing the game until a terminal node is reached, and returning the game value which is then used for updating node information. For this paper we choose UCT (Upper Confidence bounds applied to Trees, [51]) as a policy for node selection and updating because it has been proven



(a) The selection function is applied recursively until a node with un-expanded children is found

(b) One (or more) leaf nodes are created

(c) One (or more) simulated game(s) is played

(d) The game result is propagated up the tree

Figure 3.1: How Monte Carlo tree search works.

quite successful in computer game playing [99].

As with most search algorithms, the more time MCTS spends on selecting a move, the greater the playing strength. Naturally, after searching the whole game tree, there are no further gains. However, games such as Go which are characterized by a very large branching factor have large game trees that cannot be fully searched in a feasible amount of time. Parallelizing MCTS has led to stronger game play in computer programs [17], and state of the art UCT implementations use distributed algorithms which scale well on thousands of cores [115]. Unfortunately, the prohibitive cost of highly parallel machines has limited the full exploration of the potential of these algorithms.

However, a new type of massively parallel processors capable of running thousands of threads in Single Instruction Multiple Thread (SIMT) fashion, with performance in the Teraflops range, has become mainstream. These processors, called Graphics Processing Units (GPUs), are widely available on most of the current computer systems, ranging from smartphones to supercomputers. So far, there have been only a few attempts of harnessing this computing power for heuristic search.

In the remainder of the paper we first describe the traditional MCTS parallelization techniques on multi-core CPUs and computer clusters, which is followed by an introduction of GPU architectures and an existing *Block Parallel* MCTS algorithm designed to take advantage of this hardware. We then propose two new algorithms to improve on *Block Parallel* and describe implementation details. After discussing experimental results, we finish the paper with a concluding section that also suggests future work in this area.

3.2 Background and Related Work

In addition to its success when applied to board games, MCTS is easier to parallelize than $\alpha\beta$ [14, 39, 13, 49, 17]. There are several existing methods for parallelizing MCTS, which we describe in the following subsections. We start by describing CPU only parallel algorithms and then present advantages and disadvantages of GPU parallelization followed by describing an MCTS algorithm which makes use of both CPU and GPU parallelization which we we will improve upon.

3.2.1 CPU Only Parallel MCTS

The most common methods for parallelizing MCTS have been classified by [17] as root parallelization (Figure 3.2), leaf parallelization and tree parallelization (Figure 3.3). Leaf parallelization is the easiest to implement, but it usually has the worst performance. It works with a single tree and each time a node is expanded, several parallel playouts are performed, with the goal of getting a more accurate value for the node. Root parallelization constructs several

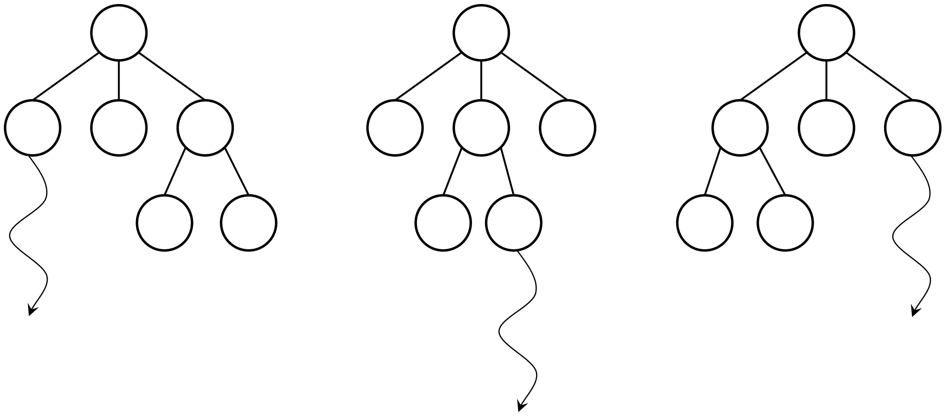


Figure 3.2: Root Parallel MCTS.

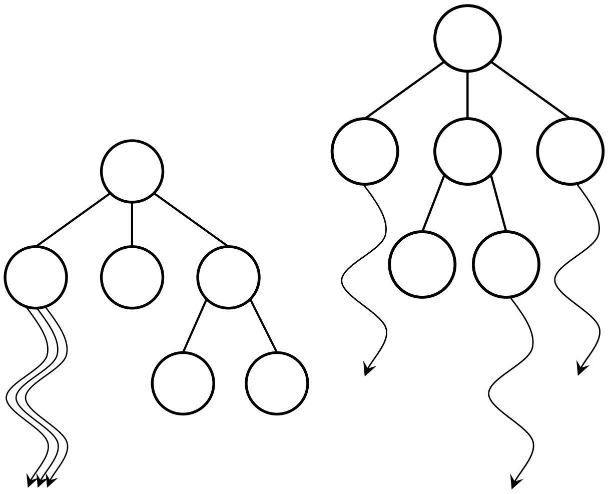


Figure 3.3: Leaf Parallel and Tree Parallel MCTS.

independent search trees, and combines them at the end. It has the least communication overhead, which makes it well suited for message passing parallel machines, for which it has shown good results. Tree parallelization works by sharing a single tree among different threads, with access coordinated by either a single global mutex, or several local mutexes. It works best on shared memory systems and its performance is highly dependent on the ratio of time spent in-tree to time spent on playouts.

3.2.2 GPU Hardware for Parallelization

In 2006 NVIDIA launched its GeForce 8800 videocard with the G80 GPU, which at the time was a major leap forward both in terms of graphics performance and in architecture. It was the first card to allow full *General-Purpose* *computing on Graphics Processing Units* (GPGPU) through its *Compute Unified Device Architecture* (CUDA) programming platform. This new architecture has since then evolved into processors capable of executing thousands of threads in parallel, using simple cores called *Streaming Processors* (SPs) which are grouped into several *Multiprocessors* (MPs). These cores are optimized for throughput, running as many threads as possible, but not focusing on the speed and latency of any individual thread. On the other hand, a CPU runs a small number of parallel threads, but focuses on running them as fast as possible while assuring each thread a fair share of processing time. A modern CPU is an *out-of-order superscalar* processor with *branch prediction* [57]:

- it reorders instructions to make use of instruction cycles that would otherwise be wasted,

- it provides instruction-level parallelism, executing more than one instruction during a clock cycle by simultaneously dispatching multiple instructions to redundant functional units on the processor,

- and it improves the flow in the instruction pipeline by trying to predict the outcome of the next conditional jump instruction.

A GPU has none of those features, making each thread run much more slowly.

The keys to the vast GPU processing power are a couple of design decisions: *Single Instruction Multiple Thread* (SIMT) architecture, in which a group of threads, called a Warp (presently comprised of 32 threads), all execute the same instruction. If there is control divergence within the threads in a Warp, some threads will be deactivated while others execute the divergent code, and then the roles will be reversed — which effectively decreases the instruction throughput. The second feature that contributes to performance is *zero-overhead scheduling*, which, by means of the GPU having thousands of registers, can maintain tens of thousands of 'live' threads with almost no context switch cost. This allows the GPU to hide global memory latency by switching to another thread group when one thread group requests a global memory access. The number of threads that can execute concurrently is given by the number of SPs or cores a GPU has (on current hardware between the high hundreds to low thousands). The number of live threads that can be scheduled depends on the amount of shared memory (a manually controllable L1 cache) and registers all those threads need, and can go up to tens of thousands. The particular number of standby threads in an application divided by the theoretical maximum supported by a specific GPU is called the *occupancy factor*.

Finally, it is worth mentioning that the latest GPUs support multiple kernels (the CUDA name for a function executing on the GPU) running in parallel, which cannot share MPs, however. Moreover, these GPUs can have one or two copy engines, which can manage data transfers simultaneously to and from main memory to GPU memory.

These details make it difficult to estimate the performance that can be achieved by a particular GPU on a particular application.

CUDA Programming Terminology

In the CUDA programming model, a function to be executed on the GPU is called a *kernel*. A *kernel* is executed by several threads grouped in thread *blocks*. All blocks in a kernel invocation compose a *grid*.

Each thread can synchronize and communicate via shared memory with other threads in the same block, but there can be no synchronization between blocks. Blocks need to be completely independent of each other, so that they can run and be scheduled in different MPs without any communication overhead.

We can estimate how efficiently we are utilizing the GPU by calculating the occupancy. An occupancy of 100% means that based on the hardware resources available — mainly registers, cores and shared memory — and on the resources needed by each thread, the entire theoretical computing power of the GPU is used. The resources needed depend on the *kernel* to be run; the number of registers and the memory needed is provided by the CUDA compiler. Reaching 100% occupancy using some number of blocks B each with T threads doesn't imply that all $B \cdot T$ threads will be running at the same time. It means that they are all standing by and ready to be scheduled by the thread scheduler. Thus, a configuration with 50% occupancy will not necessarily run twice as long as one with full occupancy to perform the same amount of work.

3.2.3 CPU/GPU Parallel MCTS

In [75], the authors implemented a hybrid Root/Leaf Parallel MCTS with the playouts running on a GPU. Their *Block Parallel* algorithm expands several *Root Parallel* trees on the CPU, and then runs blocks of *Leaf Parallel* playouts on the GPU, as shown in Figure 3.4. The root parallel part of the algorithm is shown in Algorithm 3.1. It searches several game trees in parallel in line 2 and then combines the values of all children of the root node and selects the best move.

Algorithm 3.2 describes leaf parallel search of one tree. After selecting a node and expanding one child, a CUDA kernel is called in line 7: PLAYOUTKERNEL⋘1,threads⋙(CHILD,T). The parameters between triple angle brackets specify that this kernel will run one block, with *threads* parallel playouts on the GPU. Their results will be copied back to main memory in the following line. Then, the tree is updated, from the last expanded node up to the root.

In a CUDA kernel, the following variables are automatically defined by the system:

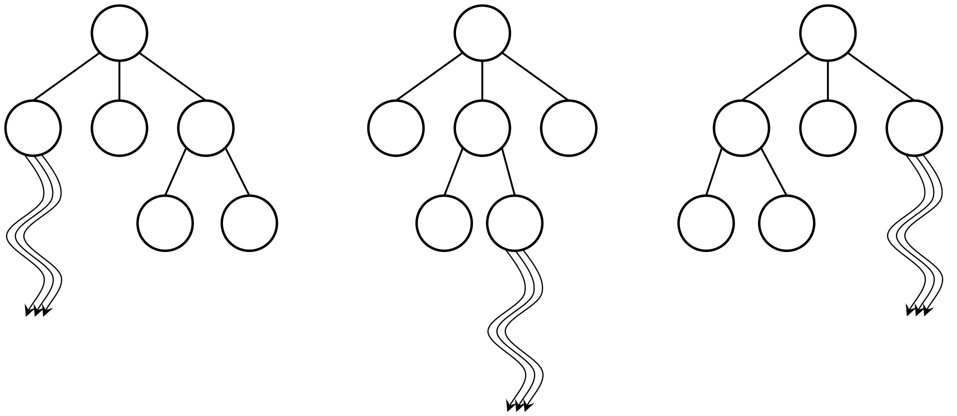


Figure 3.4: Block Parallel MCTS.

- *threadId*: the index of the current thread inside a thread block.
- *blockDim*: the number of threads in a block.
- *blockId*: the index of the current block in the grid.

The CUDA kernel shown in Algorithm 3.3 takes as parameters an array of board positions and whose turn it is to move. Each thread will run a playout for the board position indexed by its *blockId*, effectively running *blockDim* playouts for each board. Algorithm Block only uses one block to run parallel playouts for one position. Lines 3 to 12 compute the sum over the array of *blockDim* playout outcomes in parallel.

3.3 Proposed Enhancements

In this paper, we discuss the implementation of two algorithms:

1. *GPU Parallel* MCTS, which is a *Block Parallel* MCTS with the trees residing on the GPU.
2. *Multiblock Parallel* MCTS, similar to the *Block Parallel* algorithm proposed by [75], but running simulations for all the children instead of only for one child of the selected node — with the goal of fully utilizing the GPU hardware.

3.3.1 GPU Parallel

GPU Parallel is a *Block Parallel* MCTS completely executed by the GPU. After receiving the state to be searched, the GPU will construct enough independent search trees, starting from that position, to fully occupy the GPU (a few hundred are enough in our experiments). A multiple of 32 threads will be

Algorithm 3.1. Main Parallel MCTS, computes the best move for a given state.

```
function ParallelSearch(Board current, Turn t)
Require: trees                                  ▷ Number of root parallel trees
Require: time                                   ▷ Time to search
    parallel for i ← 0, trees do
        roots[i]←Block(current, t, time)
    end for
    int num←numChildren(root[0])
    for i←0, trees do
        for j←0, num do
            total[j]+=roots[i].children[j].mean
        end for
    end for
    value← $-\infty$
    for i←0, children do
        if total[i]>value then
            value←total[i]
            bestChild←roots[0].children[i].board
        end if
    end for
    bestMove←getMove(current, bestChild)
    return bestMove
end function
```

Algorithm 3.2. Block Parallel MCTS, computes a game tree for a given state.

```
function Block(Board current, Turn t, Time time)
Require: int threads                            ▷ threads per tree
    Node root(current)                          ▷ root tree at current board
    int startTime←currentTime( )
    while currentTime( )-startTime<time do
        Node node←select(root)                  ▷ UCT selection
        Node child←expand(node)                 ▷ UCT expansion
        playoutKernel⋘1,threads⋙(child,t)
        int[] wins←transfer results from GPU
        updateTree(child, wins)
    end while
    return root
end function
```

Algorithm 3.3. CUDA kernel for playouts.

```
function playoutKernel(Board[] boards, Turn t)
Require: threadId                          ▷ index of current thread
Require: blockId                           ▷ block of current thread
Require: blockDim                          ▷ threads in a block
Require: __shared__ values[blockDim]       ▷ shared array for
                                           ▷ intermediate results
    values[threadId]←randPlayout(boards[blockId])
    offset←blockDim/2
    while offset>0 do
        if threadId < offset then          ▷ parallel sum
            values[threadId] += values[threadId + offset]
        end if
        offset←offset/2
    end while
    if threadId==0 then
        wins[blockId]←values[0]
    end if
Output: int[] wins                         ▷ array with the wins for each board
end function
```

Algorithm 3.4. MultiBlock Parallel MCTS, computes a game tree for a given state.

```
function MultiBlock(Board current, Turn t, Time time)
Require: int threads                       ▷ threads per tree
    Node root(current)                     ▷ root tree at current board
    int startTime←currentTime( )
    while currentTime( )-startTime<time do
        Node node←select(root)             ▷ UCT selection
        Node[] children←expandAllChildren(node)
        int num←size(children)
        playoutKernel⋘num,threads⋙(children,t)
        int[] wins←transfer results from GPU
        for i←0, num do
            updateTree(children[i], wins[i])
        end for
    end while
    return root
end function
```

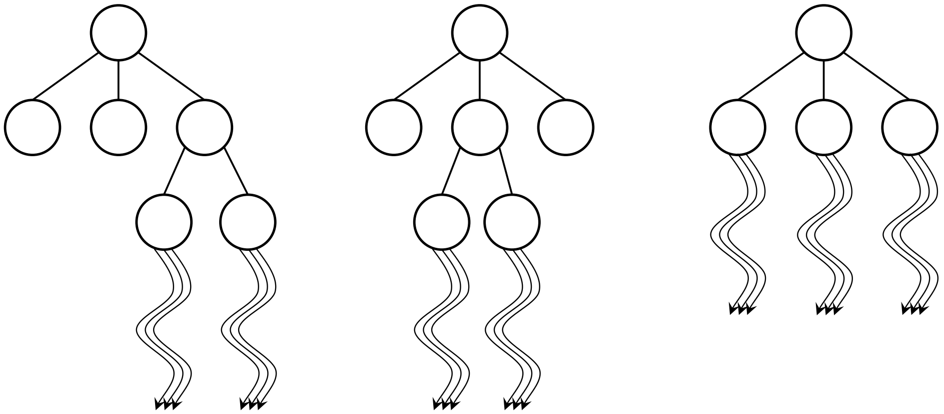


Figure 3.5: Multiblock Parallelism.

used for leaf parallelism in each of the trees. For occupancy purposes, several trees are handled by each thread block. The value of 32 threads is chosen because that is the minimum thread scheduling unit in the CUDA architecture, a *Warp*. The number of trees and threads is dependent on the specific hardware available and chosen to fully utilize the GPU.

Designing time controls for GPU processing is non-trivial because the programmer has no control of the thread scheduling which is optimized for throughput rather than latency. It is impossible to tell each thread to run for a certain amount of time because some of them may be suspended indefinitely and never get a chance to run in the allotted time. To solve this problem our program estimates in advance the number of nodes each thread should search. This estimate depends on the number of nodes per second it was able to compute during the previous move, which allows the system to adapt as the game makes a transition into easier or more difficult positions.

3.3.2 Multiblock Parallel

The *Multiblock Parallel* algorithm, shown in Figure 3.5, is quite similar to the *Block Parallel* approach proposed by [75]. However, to increase GPU occupancy, instead of performing several leaf simulations for a selected node, all the selected node's children are expanded and several simulations are performed for each of them. The algorithm in Algorithm 3.4 is quite similar to the *Block Parallel* shown in Algorithm 3.2, but in line 6 all children are expanded for the selected node, *num* stores the number of children expanded, and then the playoutKernel is called in line 8 with the array of boards corresponding to those children. The kernel is configured to run *num* blocks of *threads* threads. The number of *threads* is a parameter of the algorithm.

3.4 Experimental Results

For all experiments we used a PC with Intel Core2 Quad CPU Q8300 processor at 2.5GHz, with 8GB of RAM, running Ubuntu 12.04.2 LTS. The video card is a GeForce GTX 650 Ti, which contains a GK106 GPU with compute capability 3.0, and 2GB of RAM. This card has 4 multiprocessors running at 928Mhz, each of which can execute 192 single precision floating point operations per clock cycle. The card is therefore referred to as having 768 (4x192) CUDA Cores. Integer performance is much slower than floating point at 160 32-bit integer add/compare/logical operations per clock cycle, and 32 32-bit integer shift/multiply operations per clock cycle. As 64-bit integer operations are usually a combination of two or three 32-bit operations, the throughput is estimated to be around 200 arithmetic and 40 bitwise 64-bit integer operations per clock cycle for the combined 4 multiprocessors. The bulk of the operations in our simulations are 64-bit integer bitwise operations. This card has only one copy engine, meaning data cannot be transferred simultaneously in both directions.

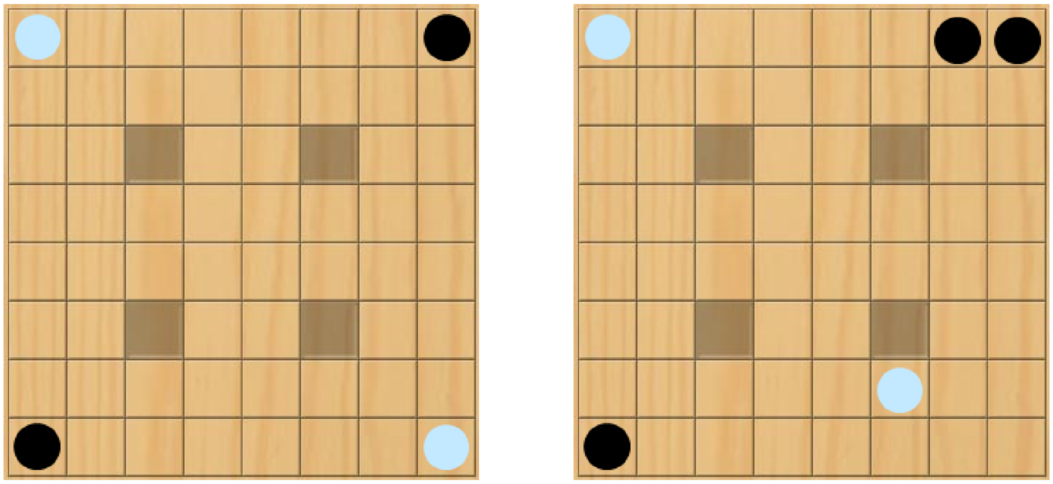
The algorithms are compared by playing 8 × 8 Ataxx. Ataxx is a 2-player, perfect information, zero-sum game. The game usually starts with two pieces in the corners of the board for each player, as shown in Figure 3.6(a). There are two type of moves in the game:

- a *clone* move adds a new piece to an empty square adjacent to any piece of the same color
- a *jump* move takes one piece and transfers it to an empty square two spaces away from its original position.

At each turn, each player must make one move as long as there are legal moves available, or pass otherwise. Figure 3.6(b) shows the board position after White made a jump move and Black responded with a clone move. It is common to have blocked squares on the board, where no pieces can be played. After each move, all opponent pieces adjacent to the moved piece are ’captured’ and switch color. The game ends when there are no more empty squares. The player with most pieces wins the game.

Table 3.1: Complexities of some games.

Game	Game-tree Complexity	State-space Complexity	Average Branching Factor	Average Game Length
Othello	10^{58}	10^{28}	10	58
Chess	10^{123}	10^{50}	35	80
Ataxx	10^{163}	10^{28}	65	90
Go	10^{360}	10^{172}	250	150



(a) Initial Ataxx board

(b) Ataxx board after one move from each player

Figure 3.6: Game of Ataxx played on a 8x8 board with 4 blocked squares.

We computed the average branching factor and Ataxx game length for the board in Figure 3.6 by running a few hundred games. We obtained an average branching factor of 65 which is almost double to that of chess, and a game length of 90 moves. These values, along with the estimated state-space complexity and game-tree complexity for Ataxx, Othello, Chess, and Go are shown in Table 3.1. For Ataxx, we estimated the state-space complexity as approximately $3^{60} \simeq 10^{28}$ (each of the 60 non-blocked squares can be empty, white or black) and the game-tree complexity around $65^{90} \simeq 10^{163}$ (average branching factor to the power of average game length). For the other games we use the values from [1].

Ataxx was chosen because it has a bigger game tree complexity than Othello — the game used to evaluate the *Block Parallel* algorithm in [75] — but is still simple enough to be easily implemented in CUDA.

All experiments were performed with 200 games per data point and 100ms per move, counting a win as 1 point, a tie as 0.5 points and a loss as 0 points. The games were played on 10 different starting configurations, with 0, 4 or 8 blocked squares. The UCT exploration constant was tuned using the same parameters.

We will assess the performance of the parallel algorithms running on the GPU by playing them against a sequential single threaded UCT running on the CPU.

All of the evaluated algorithms use the same playout function, a simple random playout until the end of the game is reached. This function is not optimized for any specific target architecture.

3.4.1 GPU Results

In this first experiment the *GPU Parallel* algorithm uses 128, 256, or 512 trees, each with 32 leaf parallel threads. Four trees are handled by each thread block. The sequential algorithm always uses one thread and one tree.

Table 3.2 shows the average number of simulations (playouts) per second, performed by each algorithm in middle-game positions (at move 30). Table 3.3 shows the speedup of *GPU Parallel* at its fastest settings (256 trees and 32 threads per tree) over sequential at a start, mid-game (move 30) and late-game (move 60) positions. The speed for 256 trees and 32 threads per tree is the highest because it is with this configuration that we get 100% occupancy of our GPU. Using 128 trees only gets us 50% occupancy, wasting some GPU resources, while using 512 trees gets us 200% occupancy having more threads than the GPU can schedule.

Table 3.2: Simulations per second, mid-game.

Sequential	**Trees×Threads**	GPU
88×10^3	**128×32**	1817×10^3
	256×32	2015×10^3
	512×32	1458×10^3

Table 3.3: Simulations per second speedup over sequential.

Game stage	GPU 256x32
Start	17.5
Mid	22.9
End	17.2

Table 3.4: Nodes per second, mid-game.

Sequential	**Trees×Threads**	GPU
88049	**128×32**	56784
	256×32	62971
	512×32	45578

Finally, Table 3.4 shows the speed of the algorithms in terms of nodes expanded per second. Note that although the number of nodes for the GPU algorithm (around $50,000$) is comparable to that of the sequential algorithm (close to $88,000$), those nodes are spread over a few hundred trees. Therefore, each tree is much smaller than the single tree expanded on the CPU. This accounts for the poor game strength results shown in Figure 3.7 which indicates the *GPU Parallel* algorithm performance for four occupancy percentages. We get 50% occupancy when using 128 trees and 32 threads per tree. The 75% mark corresponds to 192 trees. The 100% mark shows parameters for full occupancy, with 256 trees, while the 200% mark has 512 trees and twice the amount of threads that can be supported by the hardware. For this last setting, on average, half the threads are waiting idle and are not even available for scheduling. The 100% mark shows a peak performance of 42.5% winning rate. We can see a significant performance increase between 50% and 100% occupancy, and a slight drop when going to 200%. This seems encouraging as the expected performance on a GPU with more MPs should be higher.

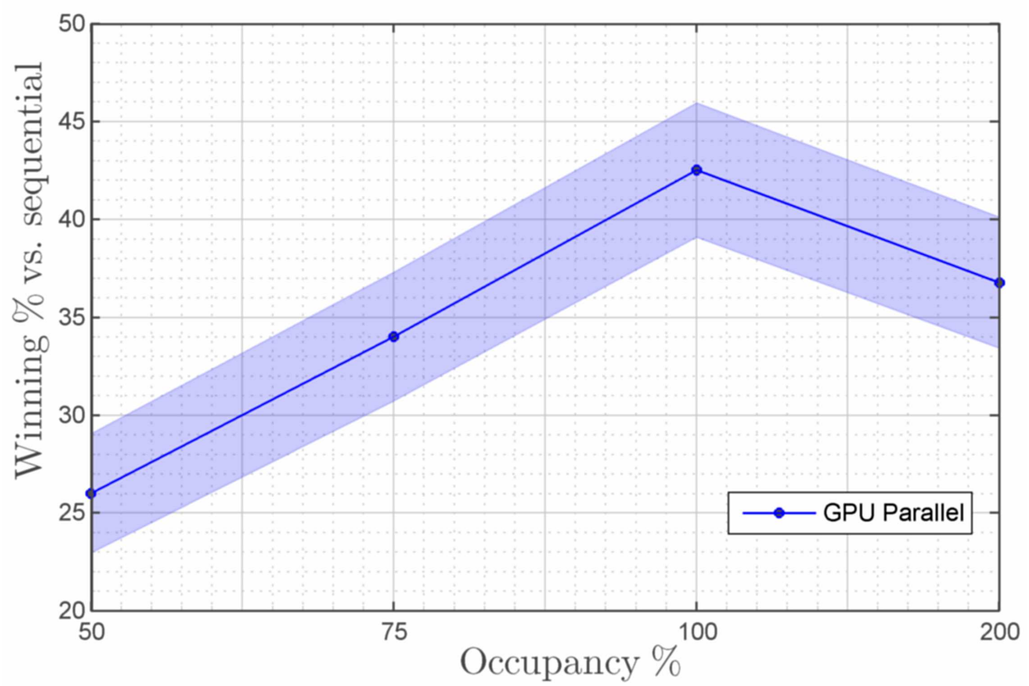


Figure 3.7: GPU Parallel winning percentage against sequential. Shaded areas represent 95% confidence intervals.

3.4.2 Multiblock Results

We implemented the *Block Parallel* and *Multiblock Parallel* algorithms with 1, 2, and 4 trees, each using 64 to 1024 GPU threads. Note that when only one tree is used, the *Block Parallel* algorithm degenerates into leaf parallelization.

Figure 3.8 shows the win percentages of different *Multiblock* configurations against the sequential MCTS. The best result is obtained by the implementation using only one three. Figure 3.9 compares *Block* and *Multiblock* algorithms using this setting. For the *Block Parallel* algorithm the number of trees

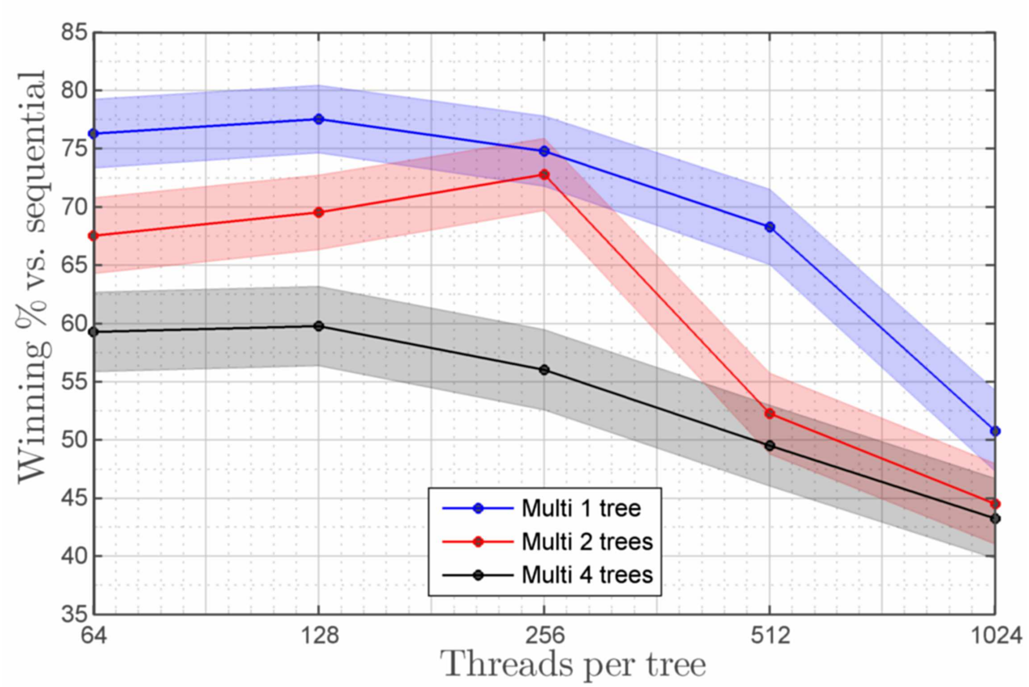


Figure 3.8: *Multiblock Parallel* winning percentage against sequential. Shaded areas represent 95% confidence intervals.

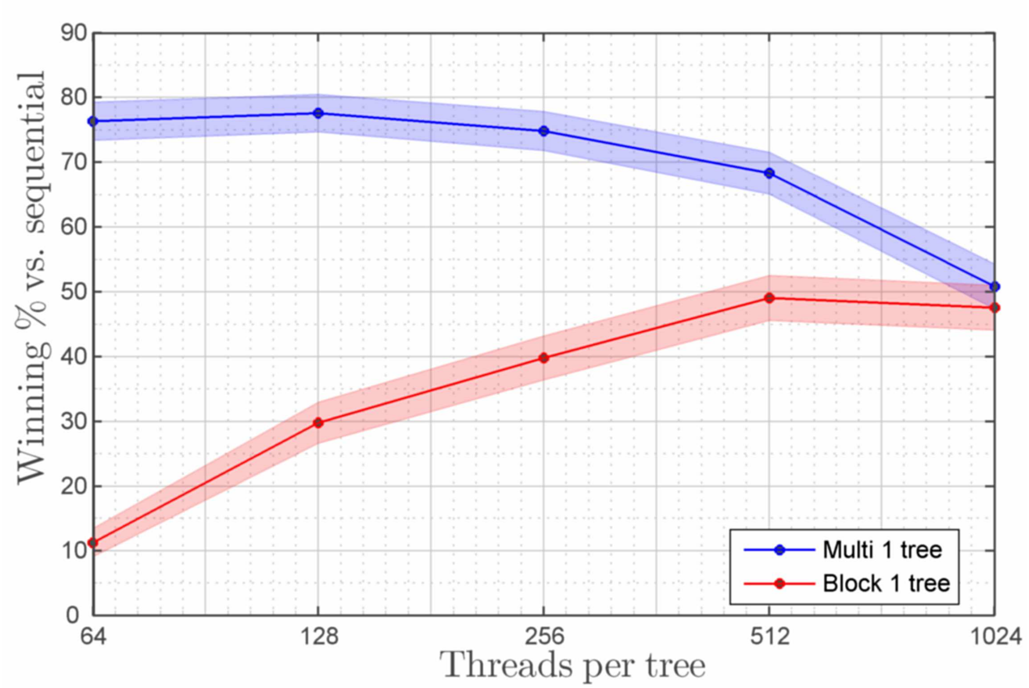


Figure 3.9: Strength comparison between *Multiblock Parallel* and *Block Parallel*. Shaded areas represent 95% confidence intervals.

didn't make a significant difference. Therefore, we used the one tree implementation for comparison. This contradicts findings by [75] which concludes that the number of trees has a bigger impact than the number threads per tree. This is likely due to the different hardware used: the Nvidia Tesla C2050 has 14 MPs (multiprocessors), each of which can execute 32 single precision floating point operations per clock cycle, for a total of 448 CUDA Cores, while the GeForce GTX 650 Ti we use has 4 multiprocessors, each of which can execute 192 single precision floating point operations per clock cycle, for a total of 768 CUDA Cores. An increased number of MPs means better parallelism for concurrent kernels, as kernels cannot share an MP. Also, the Tesla card has two copy engines, which allows it to copy data to and from the GPU memory at the same time, while the GeForce only has one copy engine, so simultaneous data transfers will be queued.

The best overall performer is the *Multiblock Parallel*, using one tree and 128 threads per node, winning 77.5% of its games. The maximum performance for the *Block Parallel* algorithm is 49.0% win rate, achieved for one tree and 512 threads.

Table 3.5: Simulations per second, mid-game.

Sequential	**Trees×Threads**	Block	Multi
88×10^3	**1×128**	106×10^3	1877×10^3
	1×256	210×10^3	2235×10^3
	1×512	376×10^3	2438×10^3

Table 3.6: Simulations per second speedup over sequential.

Game stage	Block 1x512	Multi 1x128
Start	3.7	17.4
Mid	4.3	21.3
End	4.9	13.7

Table 3.7: Nodes per second, mid-game.

Sequential	**Trees×Threads**	Block	Multi
88049	**1×128**	828	14667
	1×256	820	8729
	1×512	734	4761

Tables 3.5, 3.6 and 3.7, respectively, show the average number of simulations (playouts) per second, the speedup over the sequential algorithm, and the number of nodes expanded per second.

Block Parallel's speed in terms of nodes per second doesn't significantly change when increasing the number of threads on each tree. This is likely due to one tree not being enough to fully utilize the GPU. This means that the generated trees are of similar size, but the kernel is running more threads to perform simulations. Hence, the quality of the trees is likely improving which explains the rise in performance seen in Figure 3.9. On the other hand, *Multiblock Parallel* performs very differently. When using one tree both its speed (in terms of nodes per second) and its performance diminish when increasing the number of threads over 128. At this point, the trade-off between obtaining a better quality tree by adding more threads is offset by the time slowdown caused by expanding each node. Doubling the number of threads roughly halves the number of explored nodes per second in this case (table 3.7). This is due to *Multiblock Parallel* already being close to fully utilizing the GPU, as exhibited by the scant increase in simulations per second in table 3.5.

3.5 Conclusions and Future Work

Our experiments indicate that in 8×8 Ataxx, the *Multiblock Parallel* algorithm was able to successfully turn its simulation speed advantage into a playing strength advantage, while the *GPU Parallel* algorithm was not. A possible explanation is that because *GPU Parallel* is expanding many trees, each tree is not searched very deeply. Using fewer trees, but exploring each one in a *Tree Parallel* fashion may be worth trying. However, this approach may be difficult to implement because of the restrictions imposed by CUDA, which doesn't support mutexes, for example. However, the lock-less approach described in [34] might be applicable here.

An unexpected result is that even when seeing a speedup comparable to what Rocki et al. [75] found for the *Block Parallel* algorithm, we did not obtain a similar playing strength improvement. One reason for this could be the different application domain: Ataxx has a bigger branching factor than Othello, which leads to a shallower search (due to having to expand more siblings of each node, before expanding a child). However, for *Multiblock Parallel*, more siblings means a better GPU utilization, by scheduling more threads to execute the kernel.

Another reason is the different hardware used: their GPU had 14 multiprocessors with 32 CUDA cores each and 2 copy engines, while ours had 4 multiprocessors with 192 CUDA cores each and 1 copy engine. To investigate the true cause more experiments need to be conducted on different domains and a wider hardware range. An obvious follow-up would be to test our algorithms on Othello, which would allow for a more direct comparison with [75]. A different alternative would be to use artificial game trees with configurable parameters, which could give us broader insights on the interaction between game characteristics — like branching factor, game length and playout speed

— and hardware characteristics like number of MPs, CUDA cores and copy engines.

3.6 Contributions Breakdown and Updates Since Publication

Most of the work in this chapter was performed by Nicolas A. Barriga. Marius Stanescu helped write the published article and Michael Buro supervised the work.

More recent MCTS architectures, such as the one used by AlphaGo [81], perform rollouts on the CPU. The GPU is used to evaluate neural networks, which can be implemented very efficiently on SIMT architectures. A value network is used to complement the rollouts for the evaluation of nodes, and a policy network provides prior probabilities for newly expanded nodes.

Chapter 4

Hierarchical Adversarial Search

This chapter is a summary of work led by Marius Stanescu, in which I, Nicolas A. Barriga, and Michael Buro also participated. The original full length version was published [83] at the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE), 2014.

Despite recent advances in RTS game playing programs, no unified search approach has yet been developed for a full RTS game such as StarCraft. The research community has started to tackle the problem of global search in smaller scale RTS games [18, 76, 62]. Existing StarCraft agents rely on a combination of search and machine learning for specific sub-problems (build order optimization[22], combat [25], high-level strategy selection [92]) and hard-coded expert behaviour.

Even though the structure of most RTS AI systems is complex and comprised of many modules for unit control and strategy selection [114, 24, 97], 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.

In [83] we presented Hierarchical Adversarial Search (HAS) in which each search layer works at a different abstraction level, and each is given a goal and an abstract view of the game state. A 3-layer version of the model was implemented in SparCraft. The top layer chooses a set of objectives needed to win the game, the middle layer generates possible plans to accomplish those objectives, and the bottom layer evaluates those plans and executes them at the individual unit level.

Figure 4.1 shows a spatial decomposition of the search space, in which each unit can only interact with other close-by units, totally disregarding any interactions with units in other partitions. Independent searches can be run to evaluate the plans for each partition. For search purposes the game is advanced at the lowest level and the resulting states are abstracted back up the hierarchy. Experimental results in Figure 4.2 show that HAS outperforms state of the art algorithms such as Alpha-Beta, UCT, and Portfolio Search [25]

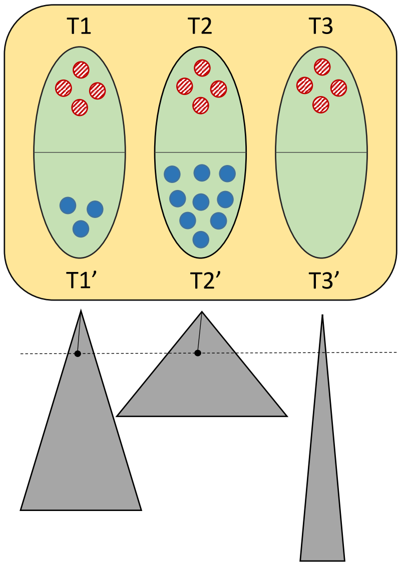
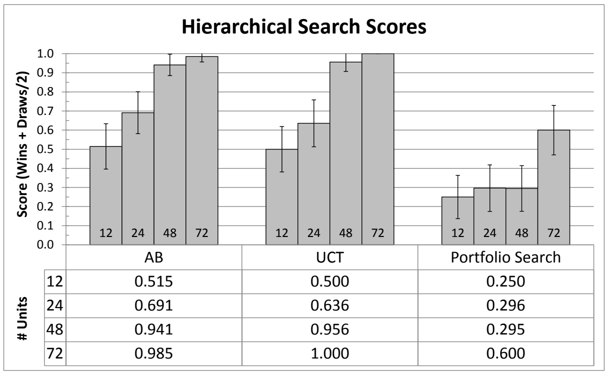


Figure 4.1: Hierarchical Adversarial Search



# 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 4.2: 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.

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.

4.1 Contributions Breakdown and Updates Since Publication

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

An alternative abstract search mechanism was presented [103, 104] at the same conference as this chapter. It involved abstracting the game state and performing look-ahead search at this abstract level, as opposed to our algorithm's forwarding of states at the lower level. No experiments were ever performed to assess the relative strengths and weaknesses of each approach.

Chapter 5

Building Placement Optimization in Real-Time Strategy Games

This chapter is joint work with Marius Stanescu and Michael Buro. It was previously published [3] 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.

Abstract

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.

5.1 Introduction

Real-Time Strategy (RTS) games are fast-paced war-simulation games which first appeared in the 1990s and enjoy great popularity ever since. RTS games pose a multitude of challenges to AI research:

- RTS games are played in real-time — by which we mean that player actions are accepted by the game engine several times per second and that game simulation proceeds even if some players elect not to act. Thus, fast to compute but non-optimal strategies may outperform optimal but compute-intensive strategies.

- RTS games are played on large maps on which large numbers of units move around under player control — collecting resources, constructing buildings, scouting, and attacking opponents with the goal of destroying all enemy buildings. This renders traditional full-width search infeasible.
- To complicate things even further, most RTS games feature the so-called "fog-of-war", whereby players' vision is limited to areas around units under their control. RTS games are therefore large-scale imperfect information games.

The initial call for AI research in RTS games [12] motivated working on RTS game AI by describing the research challenges and the great gap between human and computer playing abilities, arguing that in order to close it classic search algorithms will not suffice and proper state and action abstractions need to be developed. To this day, RTS game AI systems are still much weaker than the top human players. However, the progress achieved since the original call for research — recently surveyed in [69] — is promising. The main research thrust so far has been on tackling sub-problems such as build-order optimization, small-scale combat, and state and action inference based on analysing thousands of game transcripts. The hope is to combine these modules with high-level search to ultimately construct players able to defeat strong human players. In this paper we consider the important problem of building placement in RTS games which is concerned with constructing buildings at strategic locations with the goal of slowing down potential enemy attacks as much as possible while still allowing friendly units to move around freely.

Human expert players use optimized base layouts, whereas current programs do not and therefore become prone to devastating base attacks. For example, Figure 5.1 shows a base that is laid-out well by a human player, while Figure 5.2 depicts a rather awkward layout generated by a top StarCraft bot. The procedure we propose here assumes a given build-order and improves building locations by means of maximizing survival-related scores when being exposed to simulated attack waves whose composition has been learned from game transcripts.

In what follows we will first motivate the building placement problem further and discuss related literature. We then present our algorithm, evaluate it empirically, and finish the paper with concluding remarks and ideas for future work.

5.2 Background

Strategic building placement is crucial for top-level play in RTS games. Especially in the opening phase, player's own bases need to be protected against

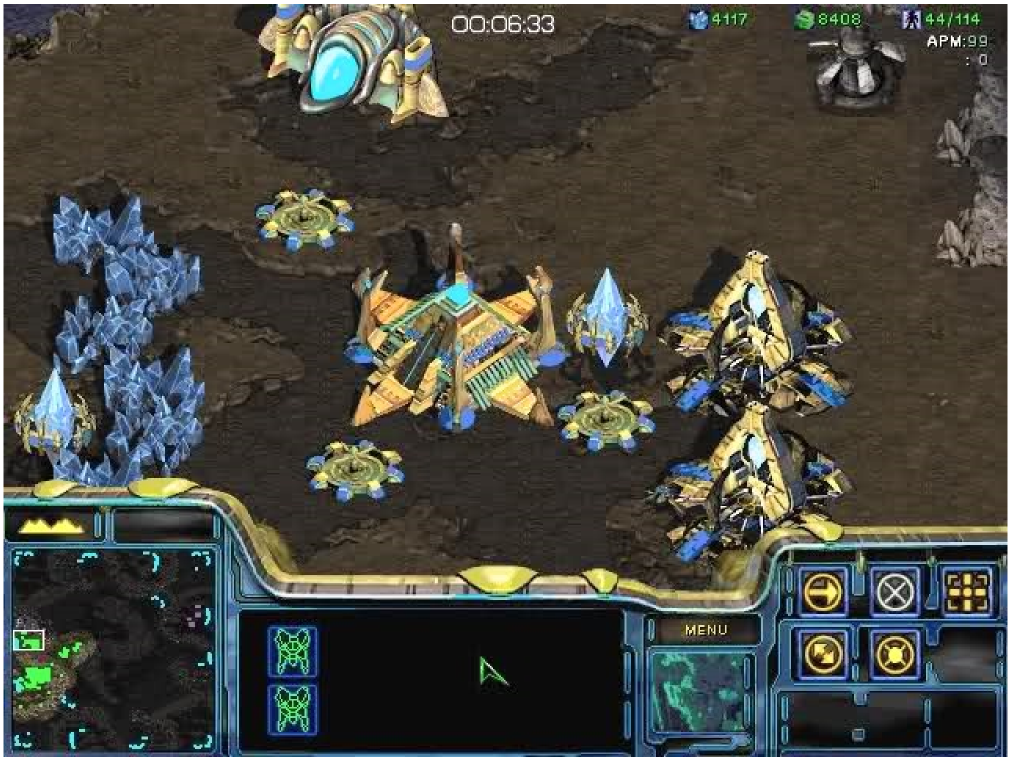


Figure 5.1: Good building placement: Structures are tightly packed and supported by cannons. (Screenshot taken from a Protoss base layout thread in the StarCraft strategy forum on TeamLiquid)

invasions by creating wall-like structures that slow opponents down so that they cannot reach resource mining workers or destroy crucial buildings. At the same time, building configurations that constrain movement of friendly units too much must be avoided. Finding good building locations is difficult. It involves both spatial and temporal reasoning, and ranges from blocking melee units completely [16] to creating bottlenecks or even maze-like configurations that maximize the time invading units are exposed to own static and mobile defenses. Important factors for particular placements are terrain features (such as ramps and the distance to expansion locations), the cost of constructing static defenses, and the type of enemy units.

Human expert players are able to optimize building locations by applying general principles such as creating choke-points, and then refining placement in the course of playing the same maps over and over and analyzing how to counter experienced opponent attacks. Methods used in state-of-the-art RTS bots are far less sophisticated [69]. Some programs utilize terrain analysis library BWTA [71] to identify chokepoints and regions to decide where to place defenses. Others simply execute pre-defined building placement scripts program authors have devised for specific maps. Still others use simple-minded spiral search around main structures to find suitable building locations. In contrast, our method — that will be described in the next section in detail —

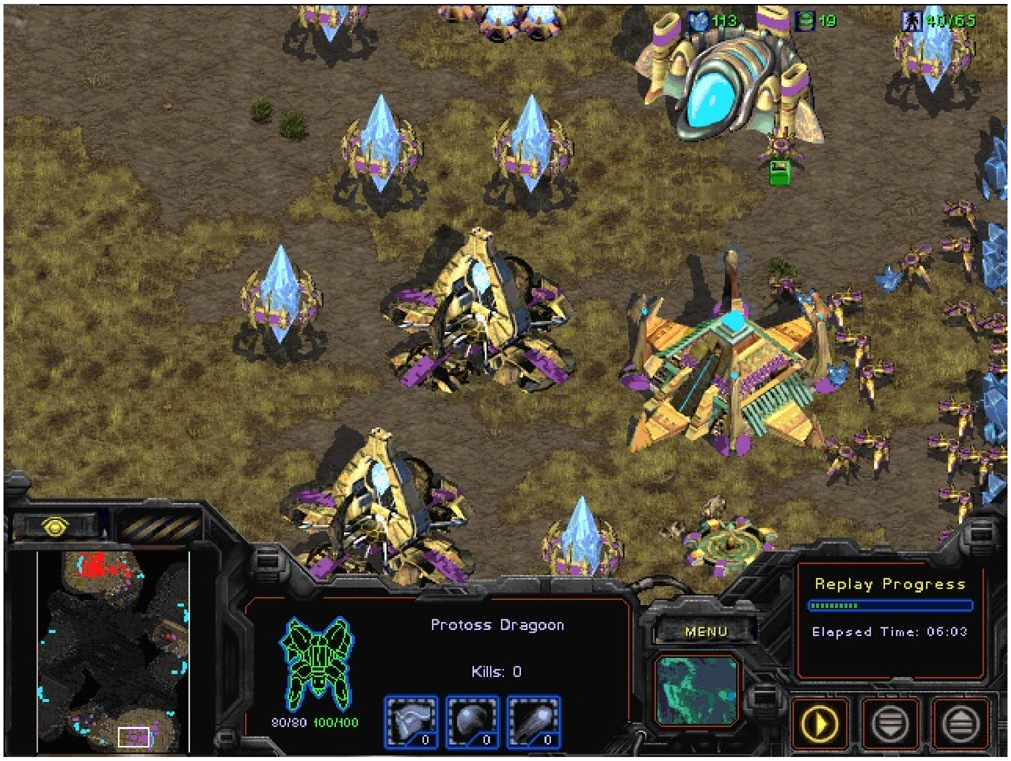


Figure 5.2: Weak building placement: structures are scattered and not well protected. (Screenshot taken from a match played by Skynet and Aiur in the 2013 AIIDE StarCraft AI competition [19].

combines fast combat simulation for gauging building placement quality with data gathered from human and bot replays for attack force estimation and stochastic hillclimbing for improving placements. The end result is a system that requires little domain knowledge and is quite flexible because the optimization is driven by an easily adjustable objective function and simulations rather than depending on hard-coded domain rules — described for instance in [16].

Building placement is a complex combinatorial optimization problem which can't be solved by exhaustive enumeration on today's computers. Instead, we need to resort to approximation algorithms such as simulated annealing, tabu search, and Genetic Algorithms — which allow us to improve solutions locally in an iterative fashion. In this paper we opt for Genetic Algorithms because building placement problem solutions can be easily mapped into chromosomes and mutation and crossover operations are intuitive and can be implemented efficiently. Good introductions to the subject can be found in [60] and [41]. For the purpose of understanding our building placement algorithm it suffices to know that Genetic Algorithms

- are stochastic hill-climbing procedures that

- encode solution instances into objects called chromosomes,
- maintain a pool of chromosomes which initially can be populated randomly or biased towards good initial solutions,
- generate new generations of chromosomes by random mutations and using so-called crossover operations that take two parents based on their fitness to generate offspring,
- and in each iteration remove weak performers from the chromosome pool.

Genetic Algorithms have been applied to other RTS game AI sub-problems such as unit micro-management [58], map generation [100], and build-order optimization [52].

5.3 Algorithm and Implementation

Our Genetic Algorithm (GA) takes a battle specification (described in the next paragraph) as input and optimizes the building placement. To asses the quality of a building placement we simulate battles defined by the candidate building placements and the mobile units listed in the battle specification. Our simulations are based on fast scripted unit behaviors which implement basic combat micro-management, such as moving towards an enemy when a unit is getting shot but doesn't have the attack range to shoot back, and smart No-OverKill (NOK) [27] targeting. The defending player tries to stay close to his buildings to protect them, while the attacker tries to destroy buildings if he is not attacked, or kill the defender units otherwise. Probes — which are essential for the economy — and pylons — needed for supply and for enabling other buildings — are considered high-priority targets. Retreat is not an option for the attacker in our simulation because we are interested in testing the base layout against a determined attack.

Our GA takes a battle specification from a file. This input consists of starting positions and initial health points of all units, and the frame in which each unit joined the battle to simulate reinforcements. The units are split between the defender player, who has buildings and some mobile units, and the attacking player who does not have any buildings.

This data can be obtained from assaults that happened in real games — as we do in our experiments — or it could be created by a bot by listing the buildings it intends to construct, units it plans to train, and its best guess for the composition and attack times of the enemy force.

Starting from a file describing the battle setup, a genome is created with the buildings positions to be optimized. Fixed buildings, such as a Nexus or Assimilator, and mobile units are stored separately because their positions are not subject to optimization.

We implemented our GA in C++ using GAlib 2.4.7 [108] and SparCraft [20]. GAlib is C++ library that provides the basic infrastructure needed for implementing GAs. SparCraft is a StarCraft combat simulator. We adapted the version from [20] by adding required extra functionality such as support for buildings, the ability to add units late during a battle, and basic collision tests and path-finding. In this implementation, all building locations are impassable to units and thus constrain their paths, and also possess hit points, making them a target for enemy units. The only buildings which have extra functionality are static defenses such as Protoss Photon Cannons, which can attack other units, and Pylons, which are needed to power most Protoss buildings. StarCraft [10], one of the most successful RTS games ever, has become the defacto testbed for AI research in RTS games after a C++ library was released in 2009 that allows C++ programs to interact with the game engine to play games [45].

5.3.1 Genetic Algorithm

Our GA is a generational Genetic Algorithm with non-overlapping populations and elitism of one individual. This is the *simple* Genetic Algorithm described in [41], which in every generation creates an entirely new population of descendants. The best individual from the previous generation is copied over to the new population by elitism, to preserve the best solution found so far. We use roulette wheel selection with linear scaling of the fitness. Under this scheme, the probability of an individual to get selected is proportional to its fitness. The termination condition is a fixed number of generations.

5.3.2 Genetic Representation

Each gene contains the position of a building, and an individual's chromosome is a fixed size array of genes. Order is always maintained (e.g., i-th gene always corresponds to the i-th building) to be able to relate each gene to a specific building.

5.3.3 Initialization

From a population of N individuals, the first $N-1$ individuals are initialized by randomly placing the buildings in the area originally occupied by the defender's base. A repair function is then called to fix illegal building locations by looking for legal positions moving along an outward spiral. Finally, we seed the population (the N-th individual) with the actual building locations from the battle description file. Using real layouts taken from human games is a feasible strategy, not only for our experimental scenario, but for a real bot competing in a tournament. Major tournaments are played on well known maps, for which we have access to at least several hundreds game replays each, and it is highly likely that some of those use a similar building composition as our bot. Otherwise, we can use layout information from our own (or another) bot's previous games, which we later show to produce similar results.

5.3.4 Genetic Operators

Because the order of the buildings in the chromosome does not hold any special meaning, there is no "real-world" relationship between two consecutive genes, which allows us to use *uniform crossover* rather than the more traditional N-point crossover. Each crossover operation takes two parents and produces two children by flipping a coin for each gene to decide which child gets which parent's gene. Afterwards, the same repairing routine that is used in the initialization phase is applied if the resulting individuals are illegal.

For each gene in a chromosome, the *mutation operator* with a small probability will move the associated building randomly in the vicinity of its current location. The vicinity size is a configurable parameter that was set to a 5 build tile radius for our experiments.

5.3.5 Fitness Function

Fitness functions evaluate and assign scores to each chromosome. The higher the fitness, the better solution the chromosome represents. To compute this value, we use SparCraft to simulate battles between waves of attackers and the individual's buildings plus mobile defenders. After the battle is concluded, we use the value (negative if the attackers won) of the winner's remaining army as the fitness score. For a given army, we compute its *value* using the following simple rules, created by the authors based on their StarCraft knowledge:

- the value of an undamaged unit is the sum of its mineral cost and 1.5 times its gas cost (gas is more difficult to acquire in StarCraft)
- the value of a damaged unit is proportional to its remaining health (e.g., half the health, half the value)
- values of workers are multiplied by 2 (workers are cheap to produce but very important)
- values of pylons are multiplied by 3 (buildings cannot function without them, and they increase the supply limit)
- finally, the value of the army is the sum of the values of its units and structures.

If we are simulating more than one attacker wave, the fitness is chosen as the lowest score after simulating all battles. Preliminary experiments showed that this gives a more robust building placement than taking the average over all battles.

5.4 Evaluation

We are interested in improving the building placement for StarCraft scenarios that are likely to be encountered in games involving highly skilled players. In particular, we focus on improving the buildings' location for given build orders and probable enemy attack groups. To obtain this data, we decided to use both high-level human replays [96] and replays from the top-3 bots in the last AIIDE StarCraft competition [19].

For a given replay, we first parse it and identify all base attacks, which are defined as a group of units attacking at least one building close to the other player's base. A base attack ends when the game is finished (if one player is completely eliminated) or when there was no unit attack in the last 15 seconds. We save all units present in the game during such a battle in a Boolean "adjacency" matrix A, where two units are adjacent if one attacked the other one or if at some point they were very close to each other during this battle interval (this matrix is symmetric). By multiplying this matrix repeatedly we can compute an "influence" matrix (e.g., A^2 tells us that a unit X *influenced* a unit Z if it attacked or was close to a unit Y that attacked or was close to Z). From this matrix we can read the connected components — in which any two units are connected to each other by paths — and thus we can easily separate different battles across the map and extract base assaults. We then filter or fix these battles according to several criteria, such as the defending base containing at least three buildings, and both players having only unit types that are supported by SparCraft. Another limitation is that SparCraft implements fast but inaccurate path-finding and collisions, so in a small percentage of games units can get "stuck". We eliminate these games from our data analysis, and thus can end up with different number of battles in different experiments. At this point the Protoss faction is the one with the most features supported by the simulator. We therefore focus on Protoss vs. Protoss battles in this paper. After considering all these restrictions, 57 battles from human replays and 31 from bot replays match our criteria.

To avoid having too few examples for each experiment, we ran the GA over this dataset several (2 to 4 depending on the experiment) times. The results vary because GA is a stochastic algorithm.

Each base assault has a fixed (observed) build order for the defending player and a list of attacking enemy units, which can appear at different time points during the battle. Using the GA presented in the previous section we try to improve the building placement and then simulate the base attack with SparCraft to test the new building configuration.

We found that most extracted base assaults strongly favour one of the players. In real games either the attacker has just a few units and tries to scout and destroy a few buildings and then retreat, or he already defeated most defender units in a battle that was not a base assault and then uses his material superiority to destroy the enemy base. These instances are not very useful

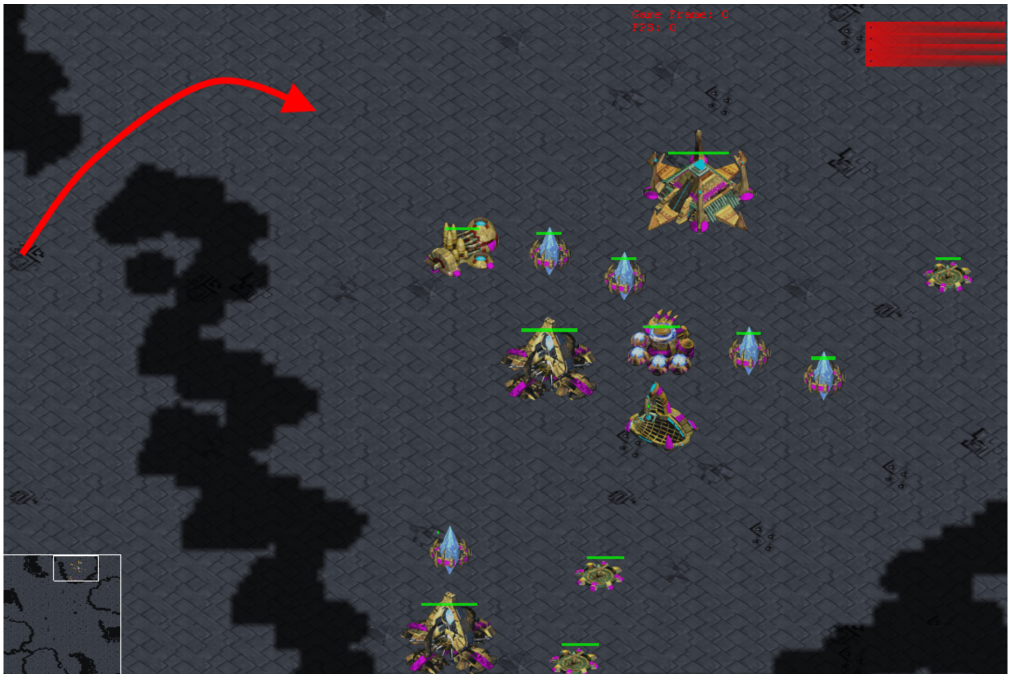


Figure 5.3: Bad building placement (created manually). The red arrow indicates the attack trajectory.

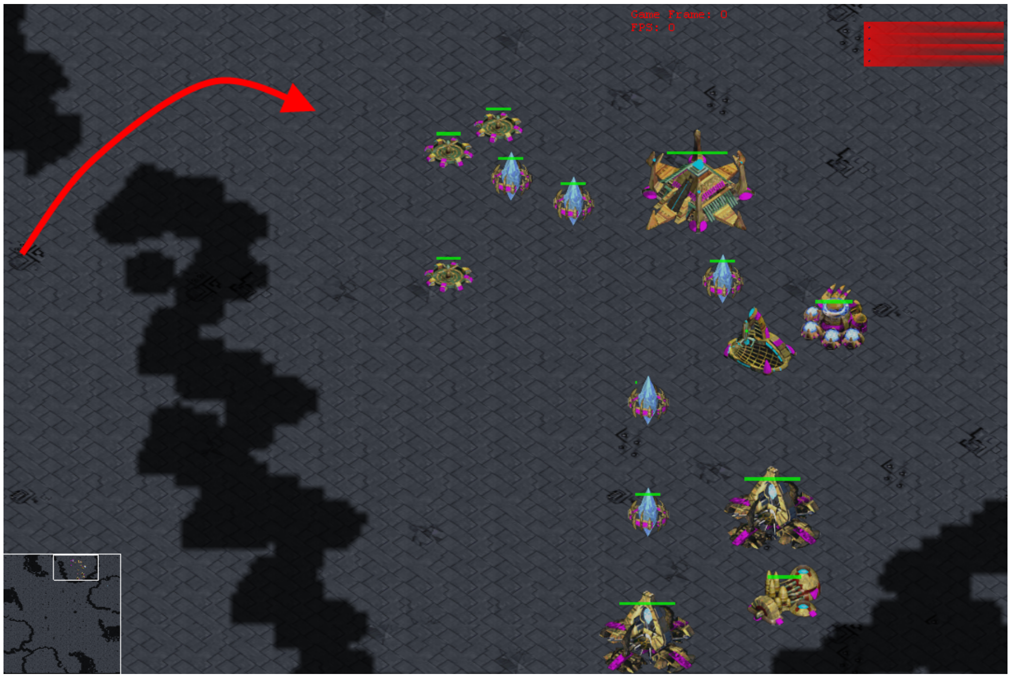


Figure 5.4: Typical improved building placement.

test cases because if the army compositions are too unbalanced, the building placement is of little importance. Consequently, to make building placement relevant, we decided to transform all games into a win for the attacker (i.e., destroying all defender units) by adding basic melee units (zealots), while keeping the build order unchanged. We believe this is the least disruptive way of balancing the armies. Additionally, it allows us to show statistics on the percentage of games lost by the defender that can be turned into a win (i.e, destroying all attacker units and keeping some buildings alive) by means of smarter building placement.

An example of a less than optimal initial Protoss base layout is shown in Figure 5.3. Figure 5.4 shows an improved layout for the same buildings, proposed by our algorithm. The attacking units follow the trajectory depicted by the red arrow.

For every base assault the algorithm performs simulations using the defender buildings and army described in the battle specification file and attack groups from all other base assaults extracted from replays on the same map and close in time to the original base attack used for training. These attack waves try to emulate the estimate a bot could have about the composition of the enemy's army, having previously played against that opponent several times. After the GA optimizes the configuration the final layout is tested against the original attack group (which was not used for training/optimizing). We call this configuration **cross-optimized**.

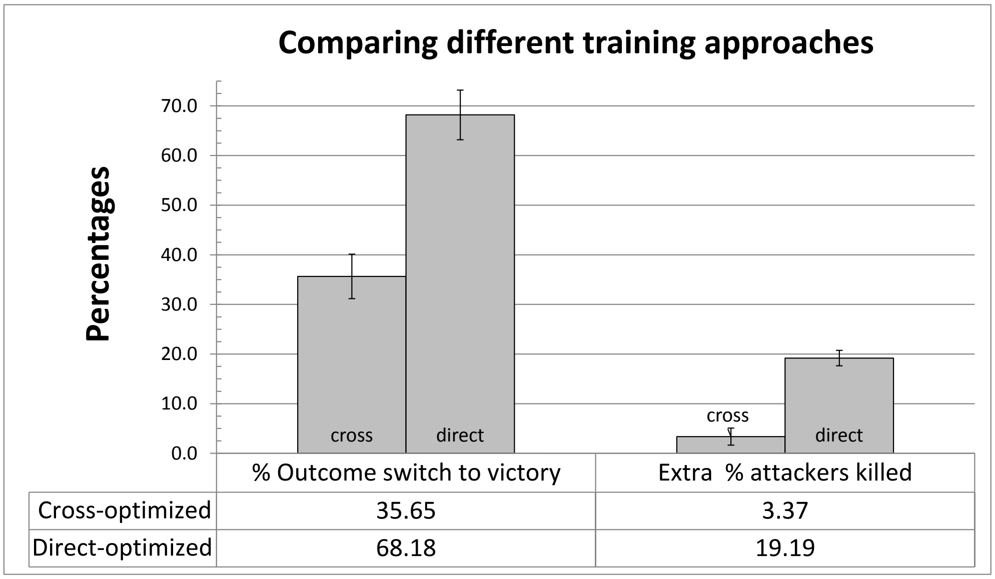
As a benchmark we also run the GA optimization against the attacker army that appeared in the actual game that we actually use for testing. This provides an upper bound estimate on the improvement we could obtain in a real game if we had a perfect estimate of the enemy's army. We call this configuration **direct-optimized**.

All experiments were performed on an Intel(R) Core2 Duo P8400 CPU 2.26GHz with 4 GB RAM running Fedora 20. The software was implemented in C++ and compiled with g++ 4.8.2 using -O3 optimization.

5.4.1 Results

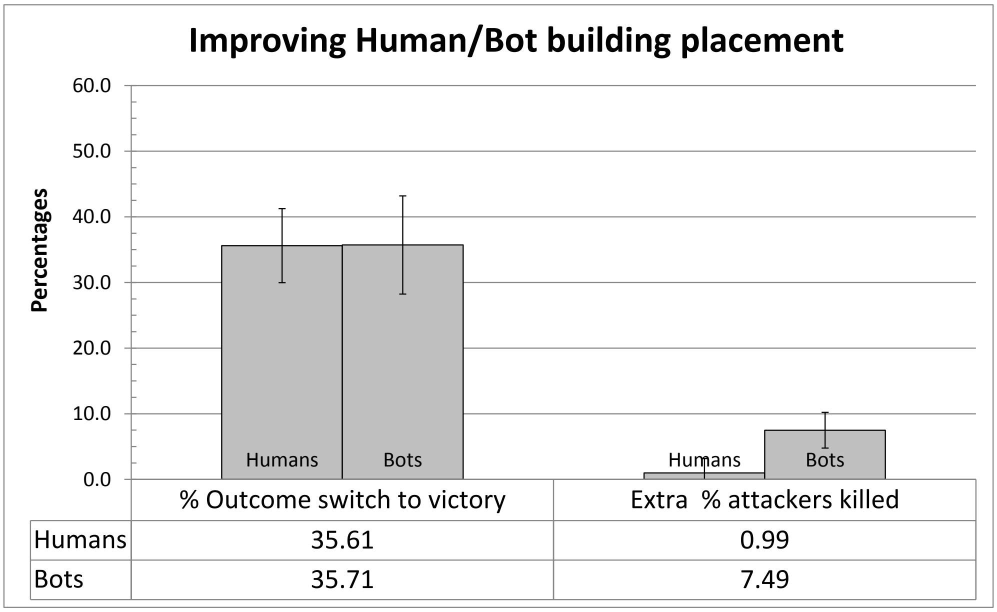
Figure 5.5 shows the improvements obtained by the GA using a population of 15 individuals and evolving for 40 generations. This might seem too little for a regular GA, but it is necessary due to the time it takes to run a simulation, and as the results show, it is sufficient. The cross-optimized GA manages to turn about a third of the battles into wins, while killing about 3% more attackers. If it had perfect knowledge of what exact attack wave to expect, represented by the direct-optimized GA, it could turn about two thirds of the battles into wins while killing about 19% more attackers.

Work on predicting the enemy's army composition would prove very valuable when combined with our algorithm. However, we are not aware of any such work, except for some geared toward identifying general strategies [110]



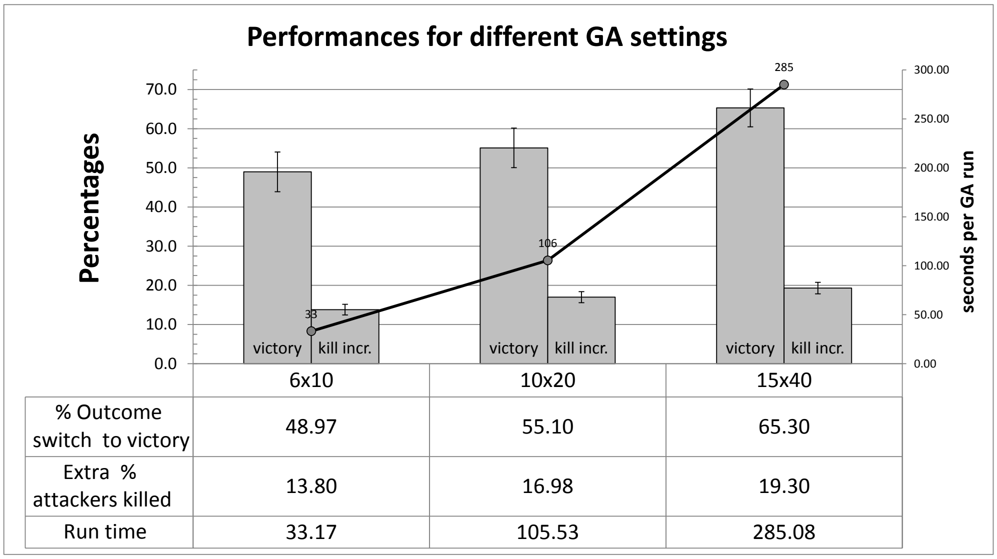
	% Outcome switch to victory	Extra % attackers killed
Cross-optimized	35.65	3.37
Direct-optimized	68.18	19.19

Figure 5.5: Percentage of losses turned into wins and extra attackers killed when cross-optimizing and direct-optimizing. 115 cross-optimized and 88 direct-optimized battles were played . Error bars show one standard error.



	% Outcome switch to victory	Extra % attackers killed
Humans	35.61	0.99
Bots	35.71	7.49

Figure 5.6: Percentage of losses turned into wins and extra attackers killed when cross-optimizing, comparing results for human and bot data. 106 human battles and 52 bot battles were played. Error bars show one standard error.



	6x10	10x20	15x40
% Outcome switch to victory	48.97	55.10	65.30
Extra % attackers killed	13.80	16.98	19.30
Run time	33.17	105.53	285.08

Figure 5.7: Percentage of losses turned into wins, extra attackers killed, and average run time for three direct-optimizing GA settings. We compare populations of 6, 10 and 15 individuals, running for 10, 20 and 40 generations respectively. 98 battles were played for each configuration. Error bars show one standard error.

or build orders [95].

Figure 5.6 compares results obtained optimizing human and bot building placements. There is some indication that bot building placement gains more from using our algorithm as more attackers are killed after optimization. However, it seems that the advantage gained is not enough to turn more defeats into victories. This result might be explained by the fact that we do not directly compare human and bot building placements, as the base assaults are always balanced such as the attacker wins before optimization. This takes away any advantage the human base layout might initially hold over ones that bots create.

Figure 5.7 shows that running longer experiments with a larger population and more generations leads to better results, as expected. A bot with a small number of pre-set build orders having access to some of his opponent's past games could improve its building placement by optimizing the building placements offline. Decent estimates for the attacking armies could be extracted from the replays and the bot could then use bigger GA parameters to obtain better placements because time is not an issue for offline training. However, Figure 5.5 shows that the best scores are attained by using accurate predictions of the enemy's army — which are more likely to be obtained during a game rather than from past replays — indicating that another possible promising approach is to use a small and fast in-game optimizer seeded with the solution from a big and slow offline optimization.

5.5 Conclusions and Future Work

We have presented a technique for optimizing building placement in RTS games that applied to StarCraft is able to help the defending player to better survive base assaults. In our experiments between a third and two thirds of the losing games are turned into wins, and even if the defender still loses games, the number of surviving attackers is reduced by 3% to almost 20% depending on our ability to estimate the attacker force. The system's performance is highly dependent on how good this estimate is, inviting some research in the area of opponent army prediction.

The proposed algorithm can easily accommodate different maps and initial building placements. We ran experiments using over 20 StarCraft maps, and base layouts taken from both human and bot games. Bot games show a slightly larger improvement after optimization, as expected. Using simulations instead of handcrafted evaluation functions ensures that this technique can be easily ported to other RTS games for which simulators are available.

We see three avenues for extending our work:

- extending the application,
- enhancing the simulations, and
- improving the optimization algorithm itself.

The application can be naturally extended by including the algorithm in a bot to optimize preset build orders against enemy armies extracted from previous replays against an enemy we expect to meet in the future. When a game starts, the closest one to our needs can be loaded and used either as is, or as a seed for online optimization. Exclusive offline optimization can work because bots don't usually perform a wide variety of build orders. Online, at roughly 30 seconds per run, can be done as long as the bot has a way of predicting the most likely enemy army.

Another possible extension is to add functionality for training against successive attack waves, arriving at different times during the build order execution. The algorithm would optimize the base layout until the first attack wave, and then consider all previous buildings as fixed. Until the next attack wave arrives, it would optimize only the positions of the buildings to be constructed. The fitness function would take into account the scores for all successive waves.

The simulations could be greatly enhanced by adding support for more advanced unit types and game mechanics, such as bunkers, flying units, spellcasters and cloaking. This would allow us to explore Terran and Zerg building placements in StarCraft, at any point in the game.

Finally, the algorithm could benefit from exploring different ways of combining the evaluation of attack waves into the fitness function. Currently the fitness is the lowest score obtained after simulating all attack waves, which led to better results than using the average. The GA could also benefit from the use of more informed operators which integrate domain knowledge, and are aware of choke-points, how to build “walls”, or how to protect the worker line.

5.6 Contributions Breakdown and Updates Since Publication

Most of the work in this chapter was performed by Nicolas A. Barriga. Marius Stanescu helped write the published article and Michael Buro supervised the work.

One further article on the subject of building placement has been published since this chapter was written. As with previous approaches it only addresses wall construction, and proposes using potential fields to build them more efficiently [30].

There has been recent work on army composition prediction [88] that could be combined with our building placement algorithm for online optimization.

Chapter 6

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

This chapter is joint work with Marius Stanescu and Michael Buro. It was previously published [5] at the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE), 2015.

Abstract

Real-Time Strategy (RTS) games have shown to be very resilient to standard adversarial tree search techniques. Recently, a few approaches to tackle their complexity have emerged that use game state or move abstractions, or both. Unfortunately, the supporting experiments were either limited to simpler RTS environments (μRTS, SparCraft) or lack testing against state-of-the-art game playing agents.

Here, we propose *Puppet Search*, a new adversarial search framework based on scripts that can expose choice points to a look-ahead search procedure. 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.

We implemented the algorithm in a complete StarCraft bot. Experiments show that it matches or outperforms all of the individual scripts that it uses when playing against state-of-the-art bots from the 2014 AIIDE StarCraft

competition.

6.1 Introduction

Unlike several abstract games such as Chess, Checkers, or Backgammon, for which strong AI systems now exist that play on par with or even defeat the best human players, progress on AI systems for Real-Time Strategy (RTS) video games has been slow [69]. For example, in the man-machine matches following the annual StarCraft AI competitions held at the AIIDE conference, a strong human player was able to defeat the best bots with ease in recent years.

When analysing these games several reasons for this playing strength gap become apparent: to start with, good human players have knowledge about strong openings and playing preferences of opponents they encountered before. They also can quickly identify and exploit non-optimal opponent behaviour, and — crucially — they are able to generate robust long-term plans, starting with multi-purpose build orders in the opening phase. Game AI systems, on the other hand, are still mostly scripted, have only modest opponent modelling abilities, and generally don't seem to be able to adapt to unforeseen circumstances well. In games with small branching factors a successful approach to overcome these issues is to use look-ahead search, i.e. simulating the effects of action sequences and choosing those that maximize the agent's utility. In this paper we present and evaluate an approach that mimics this process in video games featuring vast search spaces by reducing action choices by means of scripts that expose choice points to the look-ahead search.

6.2 Background

A few attempts have been made recently to use state and action abstraction in conjunction with adversarial search in RTS games. [83, 84] propose Hierarchical Adversarial Search in which each search layer works at a different abstraction level, and each is given a goal and an abstract view of the game state. The top layer of their three layer architecture chooses a set of objectives needed to win the game, the middle layer generates possible plans to accomplish those objectives, and the bottom layer evaluates those plans and executes them at the individual unit level. For search purposes the game is advanced at the lowest level and the resulting states are abstracted back up the hierarchy. Their algorithm was tested in SparCraft, a StarCraft simulator that only supports basic combat. The top level objectives, therefore, were restricted to destroying all opponent units while defending their own. Though this algorithm is general enough to encompass a full RTS game, only combat-related experiments were conducted.

Another proposal, by [103, 104], followed a different path, by abstracting the game state, searching at the highest abstraction level, and then translating the results back to the lower level for execution. The state representation uses a decomposition of the map into connected regions, and groups all units of the same type into squads in each region. Moves are restricted to squads, which can move to a neighboring region, attack, or stay idle. This approach, similar to the previous one, only deals with combat, but it was added to an existing StarCraft bot, so that it can play a full game. However, results were only presented for playing against the built-in AI which is much weaker than state-of-the-art bots.

6.3 Puppet Search

Our new search framework is called *Puppet Search*. At its core it is an action abstraction mechanism that, given a non-deterministic strategy, works by constantly selecting action choices that dictate how to continue the game based on look-ahead search results. Non-deterministic strategies are described by scripts that have to be able to handle all aspects of the game and may expose choice points to a search algorithm the user specifies. Such choice points mark locations in the script where alternative actions are to be considered during search, very much like non-deterministic automata that are free to execute any action listed in the transition relation. So, in a sense, *Puppet Search* works like a puppeteer who controls the limbs (choice points) of a set of puppets (scripts).

More formally, we can think of applying the *Puppet Search* idea to a game as 1) creating a new game in which move options are restricted by replacing original move choices with potentially far fewer choice points exposed by a non-deterministic script, and 2) applying a search or solution technique of our choice to the transformed game, which will depend on characteristics of the new game, such as being a perfect or imperfect information game, or a zero sum or general sum game.

Because we control the number of choice points in this process, we can tailor the resulting AI systems to meet given search time constraints. For instance, suppose we are interested in creating a fast reactive system for combat in an RTS game. In this case we will allow scripts to expose only a few carefully chosen choice points, if at all, resulting in fast searches that may sometimes miss optimal moves, but generally produce acceptable action sequences quickly. Note, that scripts exposing only a few choice points or none don't necessarily produce mediocre actions because script computations can themselves be based on (local) search or other forms of optimizations. If more time is available, our search can visit more choice points and generate better moves. Finally, for anytime decision scenarios, one can envision an iterative widening approach that over time increases the number of choice points the scripts expose, thereby improving move quality.

The idea of scripts exposing choice points originated from witnessing poor performance of scripted RTS AI systems and realizing that one possible improvement is to let look-ahead search make fundamental decisions based on evaluating the impact of chosen action sequences. Currently, RTS game AI systems still rely on scripted high level strategies [69], which, for example, may contain code that checks whether now is a good time to launch an all-in attack based on some state feature values. However, designing code that can accurately predict the winner of such an assault is tricky, and comparable to deciding whether there is a mate in k moves in Chess using static rules. In terms of code complexity and accuracy it is much preferable to launch a search to decide the issue, assuming sufficient computational resources are available. Likewise, letting look-ahead search decide which script choice point actions to take in complex video games has the potential to improve decision quality considerably while simplifying code complexity.

In the rest of the paper we will use the following terminology when discussing general aspects of *Puppet Search* and its application to RTS games in particular:

Game Move: a move that can be applied directly to the game state. It could be a simple move such as placing a stone in Go, or a combined move like instructing several units to attack somewhere while at the same time ordering the construction of a building and the research of an upgrade in an RTS game like StarCraft.

Script: a function that takes a game state and produces a game move. How the move is produced is irrelevant — it could be rule based, search based, etc. A script can expose choice points, asking the caller of the function to make a decision on each particular choice applicable to the current game state, which we call **puppet move**. For instance, one possible implementation could provide function $\textsc{getChoices}(s)$ which returns a set of puppet moves in game state s that a script offers, together with function $\textsc{execChoice}(s, m)$ that applies script moves to state s following puppet move choice m.

6.3.1 Applying Puppet Search

Reducing the search space in the fashion described above allows AI systems to evaluate long-term effects of actions, which for instance has been crucial to the success of Chess programs. If we wanted to apply *Puppet Search* to a standard two player, zero sum, perfect information game we can use any existing adversarial tree search algorithm and have it search over sequences of puppet moves. Monte-Carlo Tree Search (MCTS) [51] seems particularly well suited, as it does not require us to craft an evaluation function, and scripts already define playout policies. Alpha-Beta search could also be used, granted a suitable evaluation function can be provided for the game.

Puppet Search does not concern itself with either the origin or the inner workings of scripts. E.g., scripts can be produced by hand coding expert knowledge, or via machine learning approaches [94, 67]. Also, scripts can produce moves in a rule-based fashion or they can be search based [62, 25].

Algorithm 6.1 shows a variant of *Puppet Search* which is based on ABCD (Alpha-Beta Considering Durations) search, that itself is an adaptation of Alpha-Beta search to games with simultaneous and durative actions [27]. To reduce the computational complexity of solving multi-step simultaneous move games, ABCD search implements approximations based on move serialization policies which specify the player which is to move next (line 3) and the opponent thereafter. Policies they discuss include random, alternating, and alternating in 1-2-2-1 fashion, to even out first or second player advantages.

To fit into the *Puppet Search* framework for our hypothetical simultaneous move game we modified ABCD search so that it considers puppet move sequences and takes into account that at any point in time both players execute a puppet move. The maximum search depth is assumed to be even, which lets both players select a puppet move to forward the world in line 9. Moves for the current player are generated in line 4. They contain choice point decisions as well as the player whose move it is. Afterwards, if no move was passed from the previous recursive call (line 5), the current player's move m_2 is passed on to a subsequent PUPPETABCD call at line 6. Otherwise, both players' moves are applied to the state (line 9). The exact mechanism of applying a move is domain specific. We will give an example specific to RTS games later (Algorithm 6.2). The resulting Algorithm 6.1 is the search routine that will be used in the *Puppet Search* application to a real RTS game which we discuss next.

6.4 Puppet Search in RTS Games

This section describes the implementation of *Puppet Search* in a StarCraft game playing agent. StarCraft is a popular RTS game in which players can issue actions to all individual game units under their control — simultaneously — several times per second. Moreover, some actions are not instantaneous — they take some time to complete and their effects are sometimes randomized, and only a partial view of the game state is available to the players. Finally, the size of the playing area, the number of available units, and the number of possible actions at any given time are several orders of magnitude larger than most board games.

Our implementation tackles some of these issues by adapting standard AI algorithms, such as serializing moves in simultaneous move settings, or ignoring some issues altogether, for instance by assuming deterministic action effects and perfect information. Also, due to software limitations such as the lack of access to the engine to forward the world during the search and the unavailability of a suitable simulator, several simplifications had to be made.

Algorithm 6.1. Puppet ABCD Search

```
procedure PuppetABCD($s, h, m_1, \alpha, \beta$)
    if $h = 0$ or terminal($s$) then return eval($s$)
    toMove $\leftarrow$ playerToMove($s$, policy, $h$)
    for $m_2$ in getChoices($s$, toMove) do
        if $m_1 = \emptyset$ then
            v $\leftarrow$ PuppetABCD($s, h-1, m_2, \alpha, \beta$)
        else
            $s' \leftarrow$copy(s)
            execChoices($s', m_1, m_2$)
            v $\leftarrow$ PuppetABCD($s', h-1, \emptyset, \alpha, \beta$)
        end if
        if toMove = MAX and $v > \alpha$ then $\alpha \leftarrow v$
        if toMove = MIN and $v < \beta$ then $\beta \leftarrow v$
        if $\alpha \geq \beta$ then break
    end for
    return toMove = MAX ? $\alpha : \beta$
end procedure
```

6.4.1 Concurrent Actions and Script Combinations

So far we have presented a puppet move as a choice in a single script. In the presence of concurrent actions in RTS games, such as “send unit A there” and “send unit B there”, it might be more natural to let a set of scripts deal with units or groups independently — each of them exposing their own choice points as necessary. To fit this into the *Puppet Search* framework, we could combine all such scripts and define a single choice point whose available (concurrent) puppet moves consist of vectors of the individual scripts’ puppet moves. For example, a concurrent puppet move could be a pair of low-level puppet moves such as “take option 2 at the choice point of the script dealing with unit group A, and option 3 at the choice point of group B”, which could mean “A assaults the main base now and B builds a secondary base when the game reaches frame 7000”. The puppet move pair can then be executed for a fixed time period, or as long as it takes the search to produce an updated plan, or as long as no script encounters a choice point for which the move doesn’t contain a decision. An example of the latter could be that the enemy built the infrastructure to produce invisible units which we cannot detect with our current technology. If that possibility wasn’t encountered by the search, there is no decision made for that choice point.

In another scenario we may have access to multiple scripts that implement distinct full game strategies. For example, we could regard all programs participating in recent StarCraft AI competitions as scripts, and try to combine them into one StarCraft player based on the *Puppet Search* idea, by identifying weak spots in the individual strategies, exposing appropriate choice points, and then adding a choice point at the top that would give the AI system the option to continue executing one of the scripts for a given number of frames or until certain game events happen.

6.4.2 StarCraft Scripts

A common type of strategy in StarCraft is the *rush*: trying to build as many combat units as fast as possible, in an effort to destroy the opponent's base before he has the time to build suitable defences. This kind of strategy usually sacrifices long term economy in exchange for early military power. A range of rushes are possible, from quickly obtaining several low level units, to waiting some time to obtain a handful of high level units. We have implemented four rush strategies that fall in that spectrum: S1) Zealot rush: a fast rush with inexpensive units, S2) Dragoon rush: somewhat slower and stronger, S3) Zealot/Dragoon rush: combines the previous two, and S4) Dark Templar rush: slower, but creating more powerful units. We then combine these four strategies into one script with a single choice point at the top. We expect *Puppet Search* to be able to figure out which unit types are more suitable to assault the enemy's base, and adapt to changes — by switching unit types — in the opponent's defensive force composition during the game. The scripts share the rest of the functionality needed to satisfy the requirement that they must be able to play a full game.

6.4.3 Planning and Execution

During game play, a player receives the state of the game and can issue actions to be applied to that state by the game engine. This happens at every game frame, whether you are a human interacting with StarCraft via the GUI, or a bot receiving a game state object and returning a vector of actions. We call this the execution phase.

The vector of actions to be executed is decided in the planning phase, usually by some form of look-ahead search which requires a mechanism to see the results of applying actions to a state. A video game bot cannot use the game engine because sending a move to it would execute it in the game. Instead, it needs to be able to simulate the game. If a perfect game simulation existed, aligning planning and execution is easy, as it is the case for traditional board games, in which, for instance, the look-ahead search can know the exact outcome of advancing a pawn in a game of Chess. In the following subsections we will discuss problems arising from inaccurate simulations further.

Another important aspect is that in most turn based games planning and execution are interleaved: at each turn, the player to move takes some time to plan, and then executes a single move. That need not be the case in general, however, as the plan returned by the search could consist of several moves. As long as the moves are legal, we can continue executing the same plan. In our case, the plan returned by the planning phase is a collection of decisions in a script that can play a full game. So it can be used for execution as long as it doesn't reach a choice point for which no decision has been selected yet.

6.4.4 State Representation and Move Execution

The StarCraft game state contains every unit's state — position, hit points, energy, shields, current action, etc. — plus some general information about each side such as upgrades, resources, and map view. We will make a copy of all this information into a state data structure of our own which is used in our search procedure.

During the search, applying a puppet move 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. We use the Build Order Search System (BOSS) [23] for forwarding the economy and SparCraft [25] for forwarding battles. BOSS is a library that can simulate all the economic aspects of a StarCraft game — resource gathering, building construction, unit training and upgrades —, while SparCraft is a combat simulator. That still leaves unit movement for which we implemented a simplified version of our bot's logic during the game: during the execution phase, our bot acts according to a set of rules or heuristics to send units to either defend regions under attack or attack enemy regions. We mimic those rules, but as we cannot use the StarCraft engine to issue orders to units and observe the results of those actions, we built a high level simulation in which units are grouped into squads and these squads move from one region to another along the shortest route, towards the region they are ordered to defend or attack. If they encounter an enemy squad along the way, SparCraft is used to resolve the battle. Although this simulation is not an exact duplicate of the bot's in-game logic, it is sufficiently accurate to allow the search to evaluate move outcomes with respect to combat.

Forwarding the world by a variable number of frames generates an uneven tree in which two nodes at the same tree depth are possibly not referring to the same game time (i.e., move number in board games, or game frame in RTS games). Evaluating such nodes has to be done carefully. Unless we use an evaluation function with a global interpretation — such as winning probability or expected game score — iterative deepening by search depth cannot be performed because values of states at very different points in the game could be compared. Therefore, the iterative deepening needs to be performed with respect to game time.

Because simultaneously forwarding the economy, squads, and battles is tricky and computationally expensive, we decided that the actions that take the longest game time would dictate the pace: forwarding the economy, as shown in Algorithm 6.2. Every time we need to apply a move, the script is run, and it returns an ordered list of buildings to construct (lines 3, 4).

Algorithm 6.2. Executing Choices and Forwarding the World

```
 1: procedure EXECCHOICES(State $s$, Move $m_1$, Move $m_2$)
 2:   define constant frameLimit = N
 3:   $b_1 \leftarrow$ GETBUILDORDER($s, m_1$)
 4:   $b_2 \leftarrow$ GETBUILDORDER($s, m_2$)
 5:   while $s$.currentFrame < frameLimit and $b_1 \neq \emptyset$ and $b_2 \neq \emptyset$ and CHECK-
      CHOICEPOINTS(s) do
 6:     $b \leftarrow$ POPNEXTORDER($b_1, b_2$)          ▷ get the next order
                                                ▷ that can be executed
 7:     $t \leftarrow$ TIME(b)                         ▷ time at which order
                                                ▷ $b$ can be executed
 8:     FORWARDECONOMY($s, t$)                   ▷ gather resources,
                                                ▷ finish training units,
                                                ▷ build buildings
 9:     BOSSENQUEUEORDER($s, b$)
10:     FORWARDSQUADS($s, t$)                    ▷ movement and combat
11:   end while
12: end procedure
```

In line 8, the state is forwarded until the prerequisites (resources and other buildings) of the next building on the list are met. The building is then added to the BOSS build queue (line 9), so that next time the economy is forwarded, its construction will start. Then, at line 10, squad movement and battles are forwarded for the same number of frames. The scripts will then be consulted to check if they have hit an undecided choice point, in which case the forwarding stops. This cycle continues until either a build list becomes empty, the game ends, or a given frame limit N is reached. For simplicity we show this frame limit as a constant in line 2.

6.4.5 Search

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. We only search to even depths which ensures that each player chooses a puppet move and both moves can be applied simultaneously. Iterative deepening is done by game time, not by search depth. In every iteration we increase the depth by N frames. When applying a pair of moves, the world will be forwarded by N frames until one of the build orders is completed or until a new choice point is reached. The algorithm then continues with making the next recursive call. Due to the three different stopping conditions, a search for X frames into the future can reach nodes at different depths.

6.4.6 State Evaluation

We evaluated a few approaches, such as the destroy score used in [104, 103] or LTD2 [54, 27]. The first one is a score assigned by StarCraft to each unit based on the resources required to build it, which for our purposes overvalues buildings. The second is LTD2 ("Life-Time-Damage-2"), a measure of the average damage a unit can deal over its lifetime, which doesn't value non-attacking units and buildings at all, and even for combat units, it doesn't account for properties such as range or speed, or special abilities like cloaking. Instead of trying to handcraft an evaluation function that addresses these shortcomings we decided to use a model based on Lanchester's attrition laws presented in [85]. It automatically trains an evaluation function for units, tuned to each individual opponent. It still only accounts for combat units, so an obvious next step would be to use a function that evaluates the entire game state as described in [35].

6.4.7 Hash Tables

Because move execution is costly we use a hash table to store states, indexed by hashing the sequence of moves used to get to that state. It works similarly to a transposition table based on Zobrist hashing [116], but as the state is too big and costly to hash, we instead hash the sequence of moves leading to the current state from the root. Clearing a big hash table can be costly. So, to avoid clearing it after every search, at the root of the tree the hash value is seeded with a 64-bit random number. This makes it very unlikely that hashes from two different searches match. The table is then indexed by the hash value modulo the table size, and the hash value itself is stored along with the state. This hash table can be queried to see if a certain move was previously applied to a certain state, in which case the successor state can simply be retrieved instead of applying the move again. A standard transposition table is also used, with the same hashing mechanism as the hash table. As we are hashing the moves that lead to a state, rather than the state itself, we don't expect any transpositions to arise with our hashing mechanism, so the table is only used for retrieving best moves for states already seen which likely lead to earlier beta cut-offs when considered first.

6.4.8 Implementation Limitations

StarCraft runs at 24 frames per second, but our search needs more time than the 42[ms] between frames. StarCraft also has a hard limit of 45 seconds of inactive time before forcefully forfeiting the game. To handle this limitations, our current implementation freezes the game for six seconds of thinking time when it needs to get a new puppet move. This happens whenever the build order returned by the previous puppet move is fully constructed, approximately every 1500 to 2500 frames. We found this to be a good balance between the depth of a single search instance, vs. how many searches we can execute during a complete game. Future improvements could be either to spread the search computation across several frames, or to move it to a background thread, which is not trivial due to BWAPI not being thread safe.

Dealing with imperfect information is not in this paper's scope, so in our experiments we disable StarCraft's *Fog of War*. Having the full game state information available to our agent, the game state at the beginning of the search contains both players' units and buildings. To avoid giving our bot an unfair advantage over its opponents, we retain our base bot's scouting behaviour and do not use the extra information except for the initial state of the search.

Both simplifications have been used before. For instance, [104] pause games for up to 30 seconds every 400 frames and also disables the *Fog of War*.

6.5 Experiments and Results

Experiments were conducted using 12 VirtualBox virtual machines (VMs), each equipped with 2 cores of an Intel Xeon E5420 CPU running at 2.5GHz, and 2GB of RAM. The guest operating system was Microsoft Windows XP SP3. The StarCraft AI Tournament Manager (`github.com/davechurchill/StarcraftAITournamentManager`) was used to coordinate the matches. The default tournament timeout policy was changed to allow our bot to spend 6 seconds of search time it needs when the build queue runs out about every 2000 frames. As our bot currently contains only Protoss scripts for *Puppet Search* to use, we play against 6 of the top AIIDE 2014 Protoss bots (Ximp, Skynet, UAlbertaBot, Xelnaga, Aiur, and MooseBot) named E1...E6 on the 10 maps used in the 2014 AIIDE StarCraft AI competition. We will compare *Puppet Search*'s performance against E1...E6 with that of 4 versions of our bot playing a fixed script (S1...S4).

Table 6.1 shows the results of *Puppet Search* compared to the fixed scripts by playing against AIIDE 2014 bots. Our bot has a higher win rate than all individual scripts, except for S1 for which the mean and median performances are not statistically different from Puppet's: the Chi-squared two-tailed P values are 0.45 and 0.39 respectively). The scripts' average is the performance we can expect of a bot that plays one of the four scripts at random. The scripts' maximum value is the performance we can expect of a bot that plays the best script against each opponent. To be able to play this best response, we would need access to the opponents' bots beforehand. This is, of course, unfeasible in a real tournament. The best we could do is select a best response to last year's version of each bot. But this would likely lead to a much less robust implementation that wouldn't be able to respond to opponent's behaviour changes well.

Table 6.1 also suggests that the search could benefit from having access to more scripts: against all opponents for which at least one of the scripts defeats

Table 6.1: Win rate of individual scripts and *Puppet Search* playing against AIIDE 2014 bots E1 . . . E6. 100 games were played between each pair of bots on 10 different maps.

	Med.	Mean	E1	E2	E3	E4	E5	E6
S1	55	49.7	0	42	47	78	68	63
S2	29.5	31.5	28	31	13	42	56	19
S3	18	19.8	6	20	3	18	54	18
S4	40.5	40.8	1	72	2	84	77	9
Avg.	34.3	35.5	8.8	41.3	16.3	55.5	63.8	27.3
Max	67.5	61.8	28	72	47	84	77	63
P.S.	61	51.8	3	55	42	72	67	72

Table 6.2: Win rate of *Puppet Search* against individual scripts. 100 games were played between each pair of bots on 10 different maps.

	S1	S2	S3	S4
P.S.	44	77	99	100

Table 6.3: Probability of winning 50 or more games out of 100 against each opponent, assuming the probabilities in Table 6.1 are the true winning probabilities of each bot.

	E1	E2	E3	E4	E5	E6
S1	0	.07	.31	>.99	>.99	>.99
P.S.	<.01	.86	.07	>.99	>.99	>.99

it more than 50% of the time *Puppet Search* also defeats it more than 50% of the time. In the cases of E2 and E6, even though only 1 script performed well (S4 and S1 respectively), *Puppet Search* achieves win rates of 55% and 72%, respectively. This last result is better than any of the scripts. On the other hand, in the two instances where no script defeated the opponent (E1 and E3), the search couldn't defeat it either.

Table 6.2 shows the performance of *Puppet Search* when matched directly against each of the individual scripts it uses. *Puppet Search* defeats three of the fixed scripts by a wide margin, while the difference with S1 is not statistically significant (the Binomial P value is 0.14).

Because the average win rates of our bot and script S1 are not statistically different, Table 6.3 shows an analysis of the probabilities of winning 50 or more games, out of 100, against each opponent. This shows that using *Puppet Search* is more robust than using a single script, it has a higher probability of winning a series of games against a single random opponent. This is due to S1 successfully exploiting opponents that can't counter its fixed strategy, while using *Puppet Search* produces a bot that can win against a wider variety of opponents, but doesn't exploit the weak opponents as much.

Table 6.4: Search speed, forward length and depth reached in 6 seconds. Average over 10 games.

	Nodes/sec	Fwd. length	Depth[frames]
Early game	648.8	1984.2	10000
Midgame	505.9	941.4	6000

We should bear in mind that the implementation evaluated here has some serious limitations, such as inaccurate squad movement and combat, an evaluation function that only accounts for combat units and searching over only a small number of scripts. Furthermore, all four scripts encode similar strategies, rushes, only changing the unit type used for the rush. Because of these limitations, the average performance doesn't show improvements over the best benchmark script. However, the analysis of the results in Table 6.3 indicates some of the outcomes we can expect from a more thorough implementation: a robust algorithm that can defeat a wider range of opponents than any fixed strategy can, and one that can take advantage of any script that can counter at least a single opponent strategy.

Table 6.4 shows the nodes visited per second, average world forward length and the depth reached by the search in the 6 seconds allotted. Depth is in frames because, as mentioned earlier, we do iterative deepening by frame rather than by tree depth. We use a frame forward limit of 2000 (Algorithm 6.2, line 2), but as there are several stopping conditions, this number varies. Smaller numbers mean a deeper tree needs to be searched to reach a given frame depth. 500 nodes per second might not seem much, but at 1000 frames of game time forwarded on average for each node, we are running 500k frames per second. Looking 6000 frames ahead means that *Puppet Search* is able to evaluate action effects at least a quarter of the average game length into the future.

To the best of our knowledge, these are the first experiments conducted for high level search algorithms in RTS games against state-of-the-art AI systems. The search system over high level states shown in [104] used StarCraft's built-in AI as a benchmark. The hierarchical search system implemented by [83] used SparCraft, a StarCraft simulator limited only to the combat aspects of RTS games.

6.6 Conclusions and Future Work

We have introduced a new search framework, *Puppet Search*, that combines scripted behaviour 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 recent work on hierarchical decomposition and high-level state representation by adding look-ahead search on top of expert knowledge in the form of non-deterministic scripts. While in our experiments the average performance against all chosen opponents is similar to the best benchmark script, further analysis indicates 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 suggest that this approach is worth further consideration.

In future work we would like to remove the current limitations of our implementation, starting with the search time limits and the need to access the full map information. Some techniques have been proposed to provide inference capabilities to estimate the enemy's current state from available scouting information [111, 93, 94, 110]. Improving the performance of the current Alpha-Beta search will require a more comprehensive evaluation function. Alternatively, switching to MCTS would need more accurate combat and squad movement simulation to perform playouts. Even with the simple scripts available, *Puppet Search* manages to produce a robust player, showing it is more than just the sum of its components. However, some effort needs to be dedicated to crafting scripts that contain choice points dealing with as wide a range of situations as possible, to provide *Puppet Search* with the building blocks it needs to adapt to any scenario.

As for the general idea of *Puppet Search*, we believe it has great potential to improve decision quality in other complex domains as well in which expert knowledge in form of non-deterministic scripts is available.

6.7 Contributions Breakdown and Updates Since Publication

The bulk of the work in this chapter was performed by Nicolas A. Barriga. Marius Stanescu provided the evaluation function used by the algorithm and helped write the published article. Michael Buro supervised the work.

This first article on *Puppet Search* left a series of open questions about the impact of several design decisions:

- forward model quality
- search algorithm
- choice points design
- replan frequency

These questions, as well as others such as the effect of different map sizes, are answered in the following chapter.

Chapter 7

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

Abstract

Significant progress has been made in recent years towards stronger Real-Time Strategy (RTS) game playing agents. Some of the latest approaches have focused on enhancing standard game tree search techniques with a smart sampling of the search space, or on directly reducing this search space. However, experiments have thus far only been performed using small scenarios. We provide experimental results on the performance of these agents on increasingly larger scenarios. Our main contribution is *Puppet Search*, a new adversarial search framework that reduces the search space by using scripts that can expose choice points to a look-ahead search procedure. Selecting a combination of a script and decisions for its choice points represents an abstract move to be applied next. Such moves can be directly executed in the actual game, or in an abstract representation of the game state which can be used by an adversarial tree search algorithm. We tested *Puppet Search* in μRTS, an abstract RTS game popular within the research community, allowing us to directly compare our algorithm against state-of-the-art agents published in the last few years. We show a similar performance to other scripted and search based agents on smaller scenarios, while outperforming them on larger ones.

7.1 Introduction

In the past 20 years AI systems for abstract games such as Backgammon, Checkers, Chess, Othello, and Go have become much stronger and are now able to defeat even the best human players. Video games introduce a host of new challenges not common to abstract games. Actions in video games can usually be issued simultaneously by all players multiple times each second. Moreover, action effects are often stochastic and delayed, players only have a partial view of the game state, and the size of the playing area, the number of units available and of possible actions at any given time are several orders of magnitude larger than in most abstract games.

The rise of professional *eSports* communities enables us to seriously engage in the development of competitive AI for video games. A game played just by amateurs could look intriguingly difficult at first glance, but top players might be easily defeated by standard game AI techniques. An example of this is Arimaa [91], which was purposely designed to be difficult for computers but easy for human players. It took a decade for game playing programs to defeat the top human players, but no new AI technique was required other than those already in use in Chess and Go programs. And after a decade we still don't know if the computer players are really strong at Arimaa or no human player has a truly deep understanding of the game. In this respect a game with a large professional player base provides a more solid challenge.

One particularly interesting and popular video game class is Real-Time Strategy (RTS) games, which are real-time war simulations in which players instruct units to gather resources, build structures and armies, seek out new resource locations and opponent positions, and destroy all opponent's buildings to win the game. Typical RTS games also feature the so-called "fog of war" (which restricts player vision to vicinities of friendly units), large game maps, possibly hundreds of mobile units that have to be orchestrated, and fast-paced game play allowing players to issue multiple commands to their units per second. Popular RTS games, such as StarCraft and Company of Heroes, constitute a multi billion dollar market and are played competitively by thousands of players.

Despite the similarities between RTS games and abstract strategy games like Chess and Go, there is a big gap in state- and action-space complexity [69] that has to be overcome if we are to successfully apply traditional AI techniques such as game tree search and solving (small) imperfect information games. These challenges, in addition to good human players still outperforming the best AI systems, make RTS games a fruitful AI research target, with applications to many other complex decision domains featuring large action spaces, imperfect information, and simultaneous real-time action execution.

To evaluate the state of the art in RTS game AI, several competitions are organized yearly, such as the ones organized at the AIIDE[1] and CIG con-

[1] `http://starcraftaicompetition.com`

ferences, and SSCAI[2]. Even though bot competitions show small incremental improvements every year, strong human players continue to defeat the best bots with ease at the annual man-machine matches organized alongside AIIDE. When analyzing the games, several reasons for this playing strength gap can be identified. Good human players have knowledge about strong game openings and playing preferences of opponents they encountered before. They can also quickly identify and exploit non-optimal opponent behaviour, and — crucially — they are able to generate robust long-term plans, starting with multi-purpose build orders in the opening phase. RTS game AI systems, on the other hand, are still mostly scripted, have only modest opponent modeling abilities, and generally don't seem to be able to adapt to unforeseen circumstances well. In games with small branching factors a successful approach to overcome these issues is look-ahead search, i.e. simulating the effects of action sequences and choosing those that maximize the agent's utility. In this paper we present and evaluate an approach that mimics this process in video games featuring vast search spaces. Our algorithm uses scripts that expose choice points to the look-ahead search in order to reduce the number of action choices. This work builds on the *Puppet Search* algorithm proposed in [5]. Here we introduce new action scripts and search regimens, along with exploring different game tree search algorithms.

In the following sections we first describe the proposed algorithm in detail, then show experimental results obtained from matches played against other state-of-the-art RTS game agents, and finish the paper with discussing related work, our conclusions and motivating future work in this area.

7.2 Algorithm Details

Our new search framework is called *Puppet Search*. At its core it is an action abstraction mechanism that, given a non-deterministic strategy, works by constantly selecting action choices that dictate how to continue the game based on look-ahead search results. Non-deterministic strategies are described by scripts that have to be able to handle all aspects of the game and may expose choice points to a search algorithm the user specifies. Such choice points mark locations in the script where alternative actions are to be considered during search, very much like non-deterministic automata that are free to execute any action listed in the transition relation. So, in a sense, *Puppet Search* works like a puppeteer who controls the limbs (choice points) of a set of puppets (scripts).

More formally, we can think of applying the *Puppet Search* idea to a game as 1) creating a new game in which move options are restricted by replacing original move choices with potentially far fewer choice points exposed by a non-deterministic script, and 2) applying an AI technique of our choice to the

[2] `http://sscaitournament.com/`

transformed game to find or approximate optimal moves. The method, such as look-ahead search or finding Nash-equilibria by solving linear programs, will depend on characteristics of the new game, such as being a perfect or imperfect information game, or a zero sum or general sum game.

Because we control the number of choice points in this process, we can tailor the resulting AI systems to meet given search time constraints. For instance, suppose we are interested in creating a fast reactive system for combat in an RTS game. In this case we will allow scripts to expose only a few carefully chosen choice points, if at all, resulting in fast searches that may sometimes miss optimal moves, but generally produce acceptable action sequences quickly. Note that scripts exposing only a few choice points or none do not necessarily produce mediocre actions because script computations can themselves be based on (local) search or other forms of optimization. If more time is available for computing moves, our search can visit more choice points to generate better moves.

The idea of scripts exposing choice points originated from witnessing poor performance of scripted RTS AI systems and realizing that one possible improvement is to let look-ahead search make fundamental decisions based on evaluating the impact of chosen action sequences. Currently, RTS game AI systems still rely on scripted high level strategies [26], which, for example, may contain code that checks whether now is a good time to launch an all-in attack based on some state feature values. However, designing code that can accurately predict the winner of such an assault is challenging, and comparable to deciding whether there is a mate in k moves in Chess using static rules. In terms of code complexity and accuracy it is much more preferable to use look-ahead search to decide the issue, assuming sufficient computational resources are available. Likewise, given that currently high-level strategies are still mostly scripted, letting search decide which script choice point actions to take in complex video games has the potential to improve decision quality considerably while simplifying code complexity.

7.2.1 Scripts

For our purposes we define a script as a function that takes a game state and returns a set of actions to be performed now. The method for generating actions is immaterial: it could be a rule based player, hand coded with expert knowledge, a machine learning or search based agent, etc. The only requirement is that the method must be able to generate actions for any legal game state. As an example consider a “rush”, which is a common strategy in RTS games that tries to build as many combat units as fast as possible in an effort to destroy the opponent’s base before suitable defenses can be built. A wide range of these aggressive attacks are possible. At one extreme end, the fastest attack can be executed using only workers, which usually deal very little damage and barely have any armor. Alternatively, the attack can be delayed until

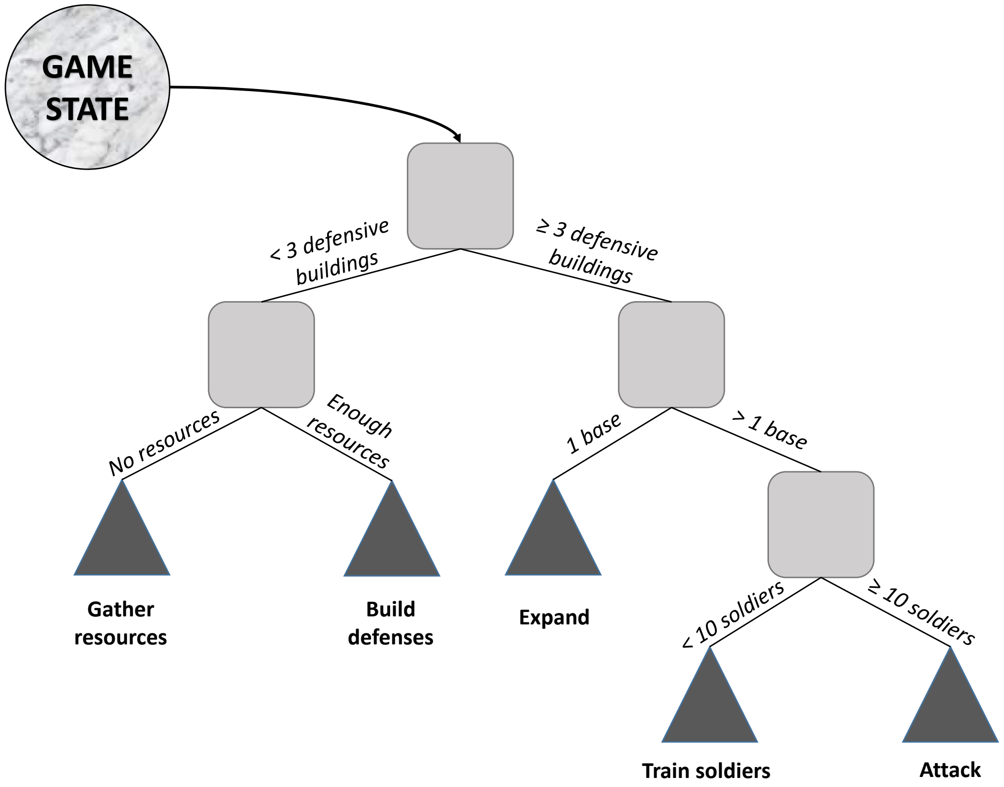


Figure 7.1: Decision tree representing script choices.

more powerful units become available.

Figure 7.1 shows a decision tree representing a script that first gathers resources, builds some defensive buildings, expands to a second resource location, creates soldiers and finally attacks the enemy. This decision tree is executed at every game simulation frame to decide what actions to issue next.

7.2.2 Choice Points

When writing a script, we must make some potentially hard choices, such as when to expand to create a new base. After training a certain number of workers, or maybe only after most of the current bases resources are depleted. Regardless of the decision, it will be hardcoded in the script, according to a set of static rules about the state of the game. Discovering predictable patterns in the way the AI acts might be frustrating for all but beginner players. Whether the behavior implemented is sensible or not in the given situation, opponents will quickly learn to exploit it and the game will likely lose some of its replay value in the process. As script writers, we would like to be able to leave some choices open, such as which units to rush with. But the script also needs to deal with any and all possible events happening during the strategy execution. The base might be attacked before it is ready to launch its own attack, or maybe the base is undefended while our infantry units are out looking for the enemy. Should they continue in hope of destroying their base before they raze ours? Or should they come back to defend? What if when we arrive at the enemy's base we realize we don't have the strength to destroy it? Should we push on nonetheless? Some, or all, of these decisions are best left open, so that they can be explored dynamically and the most appropriate choice taken during the game. We call such non-deterministic script parts choice points.

7.2.3 Using Non-Deterministic Scripts

Scripts with choice points can transform a given complex game with a large action space into a smaller games to which standard solution methods can be applied, such as learning policies using machine learning (ML) techniques, MiniMax or Monte Carlo Tree Search, or approximating Nash-equilibria in imperfect information games using linear programming or iterative methods. The number of choice points and the available choices are configurable parameters that determine the strength and speed of the resulting AI system. Fewer options will produce faster but more predictable AI systems which are suitable for beginner players, while increasing their number will lead to stronger players, at the expense of increased computational work.

ML-based agents rely on a function that takes the current game state as input, and produces a decision for each choice in the script. The parameters of that function would then be optimized either by supervised learning methods on a set of game traces, or by reinforcement learning via self-play. Once the parameters are learned, the model acts like a static (but possibly stochastic) rule based system. If the system is allowed to keep learning after the game has shipped, then there are no guarantees as to how it will evolve, possibly leading to unwanted behavior. Another approach, look-ahead search, involves executing action sequences and evaluating their outcomes. Both methods can work well. It is possible to have an unbeatable ML player if the features and training data are good enough, as well as a perfect search based player if we explore the full search space. In practice, neither requirement is easy to meet: good representations are hard to design, and time constraints prevent covering the search space in most games. Good practical results are often achieved by combining both approaches [81].

7.2.4 Game Tree Search

To decide which choice to take, we can execute a script for a given timespan, look at the resulting state, and then backtrack to the original state to try other action choices, taking into account that the opponent also has choices. To keep the implementation as general as possible, we will use no explicit opponent model. We'll assume he uses the same scripts and evaluates states the same way we do.

Algorithm 7.1 shows a variant of *Puppet Search* which is based on ABCD

Algorithm 7.1. Puppet ABCD Search

```
 1: procedure PuppetABCD($state, height, prevMove, \alpha, \beta$)
 2:    player ← playerToMove($state$, policy, $height$)
 3:    if $heigth == 0$ or terminal($state$) then
 4:       return evaluate($state$, player)
 5:    end if
 6:    for $move$ in getChoices($state$, player) do
 7:       if $prevMove == \emptyset$ then
 8:          v ← PuppetABCD($state, h - 1, move, \alpha, \beta$)
 9:       else
10:          $state'$ ←copy($state$)
11:          execChoices($state', prevMove, move$)
12:          v ← PuppetABCD($state', height - 1, \emptyset, \alpha, \beta$)
13:       end if
14:       if player = MAX and $v > \alpha$ then $\alpha \leftarrow v$
15:       if player = MIN and $v < \beta$ then $\beta \leftarrow v$
16:       if $\alpha \geq \beta$ then break
17:    end for
18:    return player == MAX ? $\alpha : \beta$
19: end procedure
20:
21: #Example call:
22: #state: current game state
23: #depth: maximum search depth, has to be even
24: $value$ = PuppetABCD($state, depth, \emptyset, -\infty, \infty$)
```

(Alpha-Beta Considering Durations) search, that itself is an adaptation of alpha-beta search to games with simultaneous and durative actions [27]. To reduce the computational complexity of solving multi-step simultaneous move games, ABCD search implements approximations based on move serialization policies which specify the player which is to move next (line 2) and the opponent thereafter. Commonly used strategies include random, alternating, and alternating in 1-2-2-1 fashion, to even out first or second player advantages.

To fit into the *Puppet Search* framework for our hypothetical simultaneous move game we modified ABCD search so that it considers puppet move sequences — series of script and choice combinations — and takes into account that at any point in time both players execute a puppet move, to perform actions at every frame in the game. The maximum search depth is assumed to be even, which allows both players to select a puppet move to forward the world in line 11. Moves for the current player are generated in line 6. They contain choice point decisions as well as the player whose move it is. Afterwards, if no move was passed from the previous recursive call (line 7), the current player's move *previousMove* is passed on to a subsequent Pup-

PETABCD call at line 8. Otherwise, both players' moves are applied to the state (line 11).

RTS games are usually fast paced. In StarCraft bot competitions, for example rules allow up to 41 milliseconds computing time per simulation frame. However, as actions take some time to complete, it is often the case that the game doesn't really change between one game frame and the next one. This means that an agent's computations can be split among several consecutive frames until an action has to be issued. To take advantage of this split computation, the recursive Algorithm 7.1 has to be transformed into an iterative one, by manually managing the call stack. The details of this implementation are beyond the scope of this paper, and can be reviewed in our published code at the μRTS repository[3].

Alpha-beta search variants conduct depth-first search, in which the search depth needs to be predefined, and no solution is returned until the algorithm runs to completion. In most adversarial video games, the game proceeds even when no actions are emitted, resulting in a disadvantage if a player is not able to issue actions in a predefined time. Algorithms to play such games have to be *anytime* algorithms, that is, be able to return a valid solution even when interrupted before completion. Iterative deepening has traditionally been used to transform alpha-beta search into an anytime algorithm. It works by calling the search procedure multiple times, increasing the search depth in each call. The time lost due to repeating computations is usually very low, due to the exponential growth of game trees. However, even this minimal loss can be offset by subsequently reusing the information gained in previous iterations for move sorting.

In our implementation we reuse previous information in two ways: a *move cache* and a *hash move*. The *move cache* is a map from a state and a pair of moves (one for each player) to the resulting state. With this cache, we can greatly reduce the time it takes to forward the world, which is the slowest operation in our search algorithm. The *hash move* is a standard move ordering technique, in which the best move from a previous iteration is used first, in order to increase the chances of an earlier beta cut. Algorithm 7.2 shows an UCT version of *Puppet Search*. The node structure on line 1 contains the game state, the player to move (moves are being serialized as in ABCD), a list of the children nodes already expanded, a list of legal moves (applicable choice point combinations), and a previous move, which can be empty for the first player.

Procedure PuppetUCTCD in line 9 shows the main loop, which calls the selectLeaf procedure at line 12, which will select a leaf, expand a new node, and return it. Then a playout is performed in line 14 using a randomized policy for both players. The playout is cut short after a predefined number of frames, and an evaluation function is used. Afterwards, a standard UCT update rule is applied. Finally, when the computation time is spent, the best

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

Algorithm 7.2. Puppet UCTCD Search

```
Structure Node
    state
    player
    children
    moves
    previousMove
End Structure

procedure PuppetUCTCD(state)
    root ← Node(state)
    while not timeUp do
        leaf ← selectLeaf(root)
        newState ←
         simulate(leaf.state, policy, policy, leaf.parent.player, leaf.player, EVAL_PLAYOUT_TIME)
        e ← evaluate(newState, leaf.player)
        update(leaf, e)
    end while
    return getBest(root)
end procedure

procedure selectLeaf(root)
    if size(root.children) < size(root.moves) then
        move ← next(root.moves)
        if root.previousMove == ∅ then
            node ← Node(root,move)
            append(root.children, node)
            return selectLeaf(node)
        else
            newState ←
        simulate(root.state, root.previousMove, move, root.parent.player, root.player, STEP_PLAYOUT_TIME)
            node ← Node(newState)
            append(root.children, node)
            return node
        end if
    else
        node ← UCB1(root.children)
        return selectLeaf(node)
    end if
end procedure
```

move at the root is selected and returned.

Procedure selectLeaf in line 21 either expands a new sibling of a leaf node, or if there are none, uses a standard UCB1 algorithm to select a node to follow down the tree. When expanding a new node, it always expands two levels, because playouts can only be performed on nodes for which both players have selected moves.

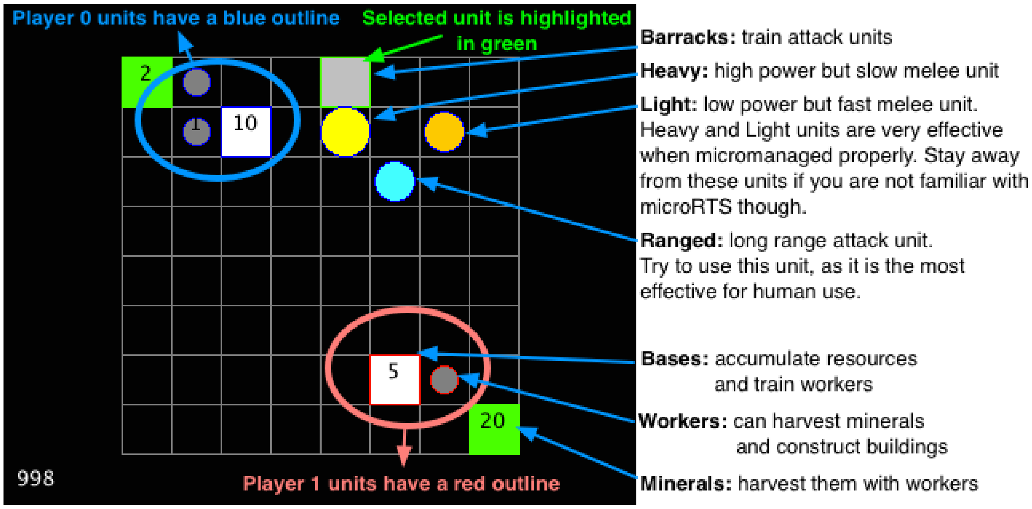
7.2.5 State Evaluation

Forwarding the state using different choices is only useful if we can evaluate the merit of the resulting states. We need to decide which of those states is more desirable from the point of view of the player performing the search. In other words, we need to evaluate those states, assign each a numerical value and use it to compare them. In zero-sum games it is sufficient to consider symmetric evaluation functions $eval(state, player)$ that return positive values for the winning player and negative values for the losing player with $eval(state, p1) = -eval(state, p2)$.

The most common approach to state evaluation in RTS games, and the one we use in our experiments, is to use a linear function that adds a set of values that are each multiplied by a weight. The values usually represent simple features, such as the number of units of each type a player has, with different weights reflecting their estimated worth. Weights can be either hand-tuned or learned from records of past games using logistic regression or similar methods. An example of a popular metric in RTS games is Life-Time Damage, or LTD [54], which tries to estimate the amount of damage a unit could deal to the enemy during its lifetime. Another feature could be the cost of building a unit, which takes advantage of the game balancing already performed by the game designers. Costlier units are highly likely to be more useful, thus the player that has a higher total unit cost has a better chance of winning. [85] describes a state-of-the-art evaluation method based on Lanchester's attrition laws that takes into account combat unit types and their health.

[35] presents a global evaluation function that takes into account not just military features, but economic, spatial and player skill as well. [87] implements a Convolutional Neural Network for game state evaluation that provides a significant increase in accuracy compared to other methods, at the cost of execution speed. How much the trade-off is worth depends on the search algorithm using it.

A somewhat different state evaluation method involves Monte Carlo simulations. Instead of invoking a static function, one could have a pair of fast scripts, either deterministic or randomized, play out the remainder of the game, and assign a positive score to the winning player [27]. The rationale behind this method is that, even if the scripts are not of high quality, as both players are using the same policy, it is likely that whoever wins more simulations is the one that was ahead in the first place. If running a simulation until the end



Player 0 units have a blue outline

Selected unit is highlighted in green

Player 1 units have a red outline

Barracks: train attack units

Heavy: high power but slow melee unit

Light: low power but fast melee unit. Heavy and Light units are very effective when micromanaged properly. Stay away from these units if you are not familiar with microRTS though.

Ranged: long range attack unit. Try to use this unit, as it is the most effective for human use.

Bases: accumulate resources and train workers

Workers: can harvest minerals and construct buildings

Minerals: harvest them with workers

Figure 7.2: Screenshot of μRTS, with explanations of the different in-game symbols.

of the game is infeasible, a hybrid method can be used that performs a limited playout for a predetermined amount of frames, and then calls the evaluation function. Evaluation functions are usually more accurate closer to the end of a game, when the game outcome is easier to predict. Therefore, moving the application of the evaluation function to the end of the playout often results in a more accurate assessment of the value of the game state.

7.2.6 Long Term Plans

RTS game states tend to change gradually, due to actions taking several frames to execute. To take advantage of this slow rate of change, we assume that the game state doesn't change for a predefined amount of time and try to perform a deeper search than otherwise possible during a single frame. We can then use the generated solution (a series of choices for a script's choice points) to control the game playing agent, while the search produces the next solution. We call this approach having a *standing plan*.

Experiments reported in the following section investigate how advantageous is the trade-off between computation time and recency of the data being used to inform the search.

7.3 Experiments and Results

The experiments reported below were performed in μRTS[4], an abstract RTS game implemented in Java and designed to test AI techniques. It provides the core features of RTS games, while keeping things as simple as possible.

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

In the basic setup only four unit and two building types are supported (all of them occupying one map tile), and there is only one resource type. μRTS is a 2-player real-time game featuring simultaneous and durative actions, possibly large maps (although sizes of 8x8 to 16x16 are most common), and by default all state variables are fully observable. μRTS comes with a few basic scripted players, as well as search-based players that implement several state-of-the-art RTS game search techniques. This makes it an useful tool for benchmarking new algorithms. Figure 7.2 shows an annotated screenshot.

μRTS comes with four scripted players, each implementing a rush strategy with different unit types. A rush is a simple strategy in which long term economic development is sacrificed for a quick army buildup and early assault on the opponent's base. The first *Puppet Search* version we tested includes a single choice point to select among these 4 existing scripts, generating a game tree with constant branching factor 4. We call this version *PuppetSingle*. A slightly more complex script was implemented as well. In addition to the 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. Both ABCD and UCTCD search were used, with and without a standing plan, leading to a total of eight different agents.

The algorithms used as a benchmark are NaïveMCTS and two versions of AHTNs: AHTN-F, the strongest one on small maps, and AHTN-P, the more scalable version. These algorithms are described in detail in section 7.4.

All experiments were conducted on computers equipped with an Intel Core i7-2600 CPU @ 3.4 GHz and 8 GB of RAM. The operating system was Ubuntu 14.04.5 LTS, and Java JDK 1.8.0 was used.

μRTS was configured in a similar manner to experiments reported in previous papers. A computational budget of 100ms was given to each player between every simulation frame. In [62, 65] games ended in ties after running for 3000 game frames without establishing a winner, i.e. a player eliminating all the other player's units. On bigger maps this produces a large number of ties, so we used different cutoff thresholds according to the map sizes:

Size	8x8	16x16	24x24	64x64	>64x64
Frames	3000	4000	5000	8000	12000

Applying these tie rules produced tie percentages ranging from 1.1% to 3.9% in our experiments.

All algorithms share the same simple evaluation function, a randomized playout policy and a playout length of 100 simulation frames ([62, 65]), which were chosen to easily compare our new results with previously published results. The *Puppet Search* versions that use a standing plan have 5 seconds of planning time between plan switches, which was experimentally determined to produce the strongest agent. The UCTCD *Puppet Search* versions use exploration factor $c = 1$ for the 8x8 map, 10 for the 16x16 map, and 0.1 for all others, tuned experimentally. All other algorithms maintain their original settings from the papers in which they were introduced.

The following maps were used in the experiments. Starting positions with a single base were chosen because they are by far the most common in commercial RTS games. Unless specified, 24 different starting positions were used for each map.

8x8: an 8x8 map with each player starting with one base and one worker. This map has been used in previous papers ([62, 65]).

16x16: a 16x16 map with each player starting with one base and one worker. This map was previously used in [65].

24x24: a 24x24 map with each player starting with one base and one worker.

BloodBath: a port of a well known 64x64 StarCraft: Brood War map. 12 different starting positions were used.

AIIDE: a port of 8 of the StarCraft: Brood War maps used for the AIIDE competition[5]. 54 different starting positions were used in total.

Figures 7.3 and 7.4 show results grouped by algorithm type. The bars on the left of each group (blue) represent the average and maximum win rates of the scripts in a round robin tournament (all versus all). The middle (red) bars show the average and maximum performance of the benchmark algorithms (NaïveMCTS, AHTN-P and AHTN-F). The bars on the right of each group (yellow) show the average and maximum performance of the eight *Puppet Search* versions we implemented.

On small maps the performances of *Puppet Search* and other search based algorithms are similar. The average performance of the scripted agents is fairly low, however, on the smaller maps, there is always one competitive script (*WorkerRush* in 8x8 and 16x16, and *LightRush* in 24x24). On the bigger maps, no agent comes close to *Puppet Search*. The differences between *Puppet Search* versions are shown below in Figures 7.5 to 7.7.

Figure 7.5 shows similar performance for the UCTCD *Puppet Search* versions and the ABCD ones on small maps. On the bigger maps ABCD has a higher winrate. This uneven performance by UCTCD can be a result of MCTS algorithms being designed for games with a larger branching factor. This weakness is masked on the smaller maps because of the larger number of nodes that can be explored due to faster script execution.

Figure 7.6 shows that *PuppetSingle* clearly outperforms *PuppetBasic* on the smaller maps, while the opposite is true on the bigger ones. This exemplifies the importance of designing choice points carefully. They must be potentially useful, otherwise they are just increasing the branching factor of the search tree without providing any benefit.

[5] `http://www.starcraftaicompetition.com`

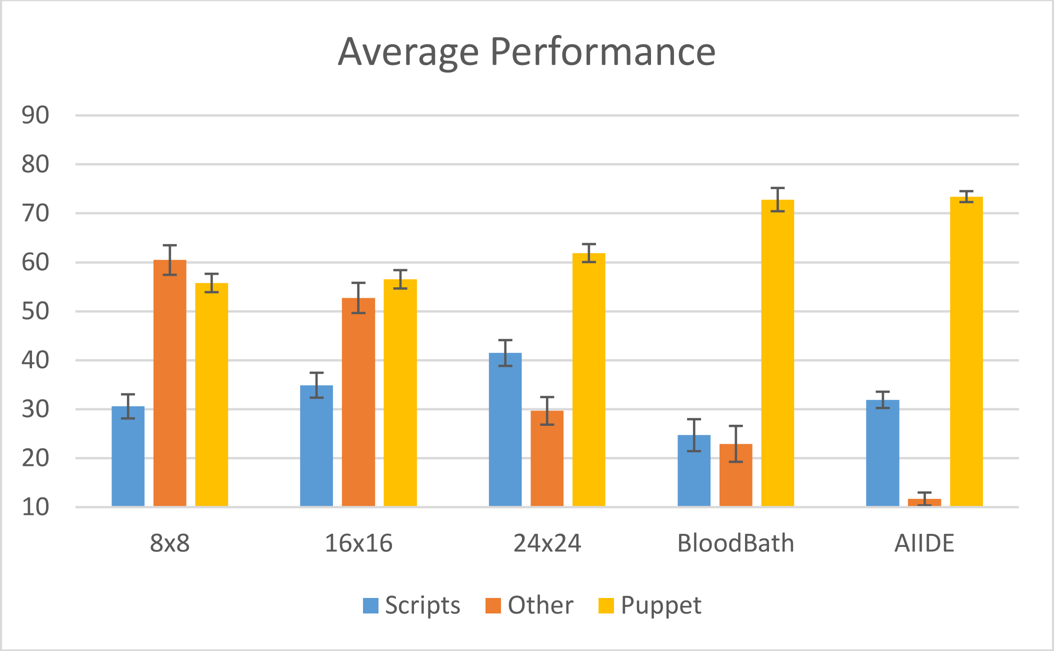


Figure 7.3: Average performance of the agents, grouped by algorithm type. Error bars indicate one standard error.

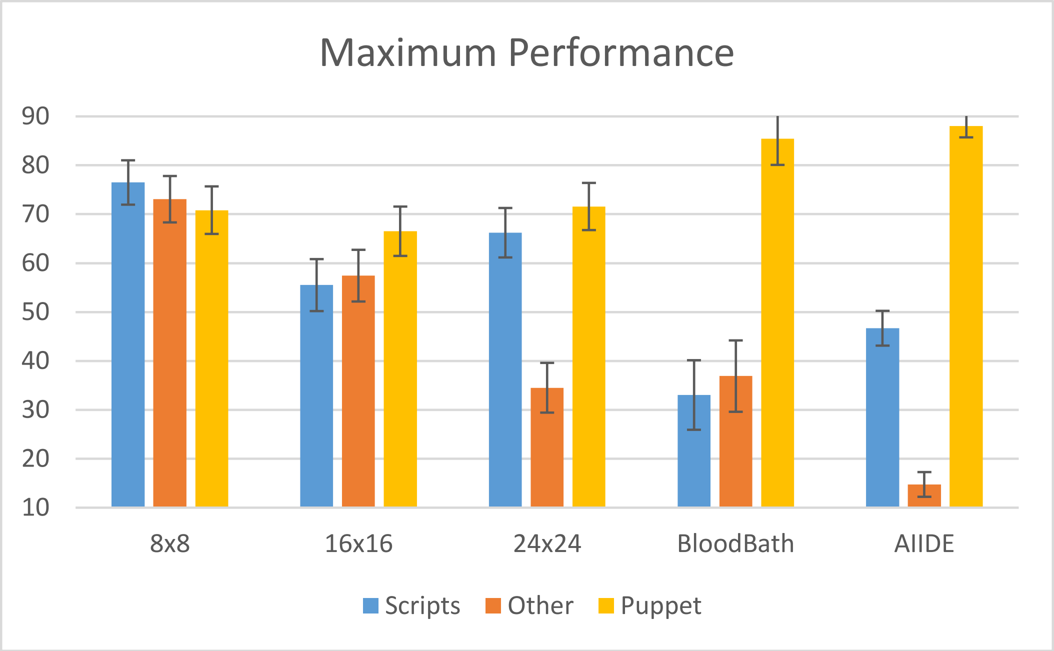


Figure 7.4: Maximum performance of each type of agent. Error bars show one standard error.

Figure 7.7 shows that investing time to compute a longer term plan, and then using it while computing the next plan, is only useful on big maps. On such maps the playouts are usually slower and the action consequences are

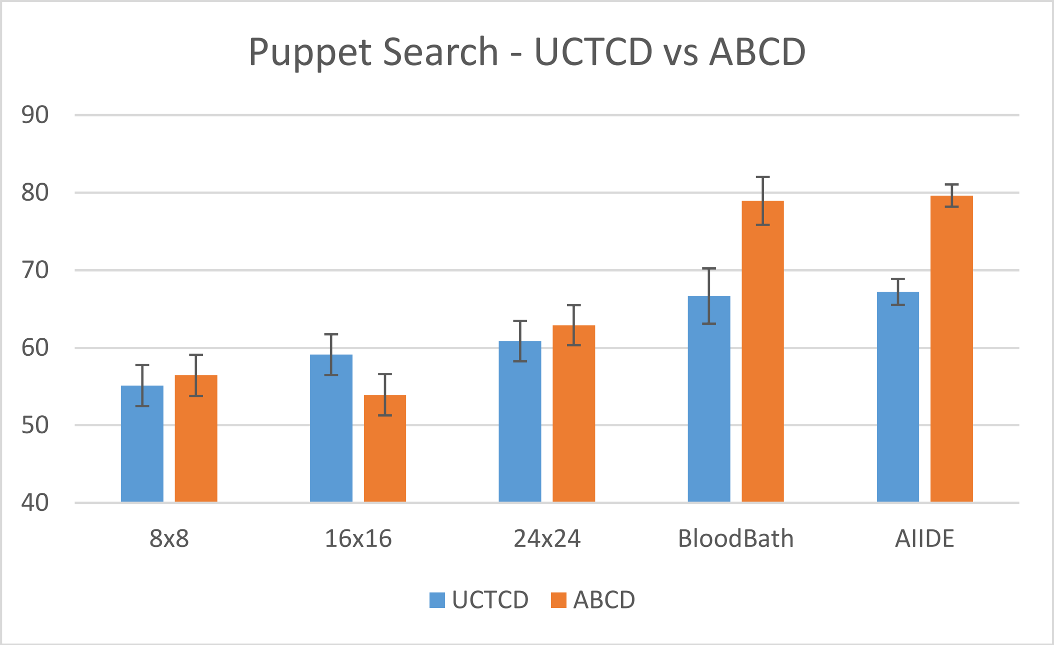


Figure 7.5: Average performance of ABCD and UCTCD versions of *Puppet Search*. Error bars show one standard error.

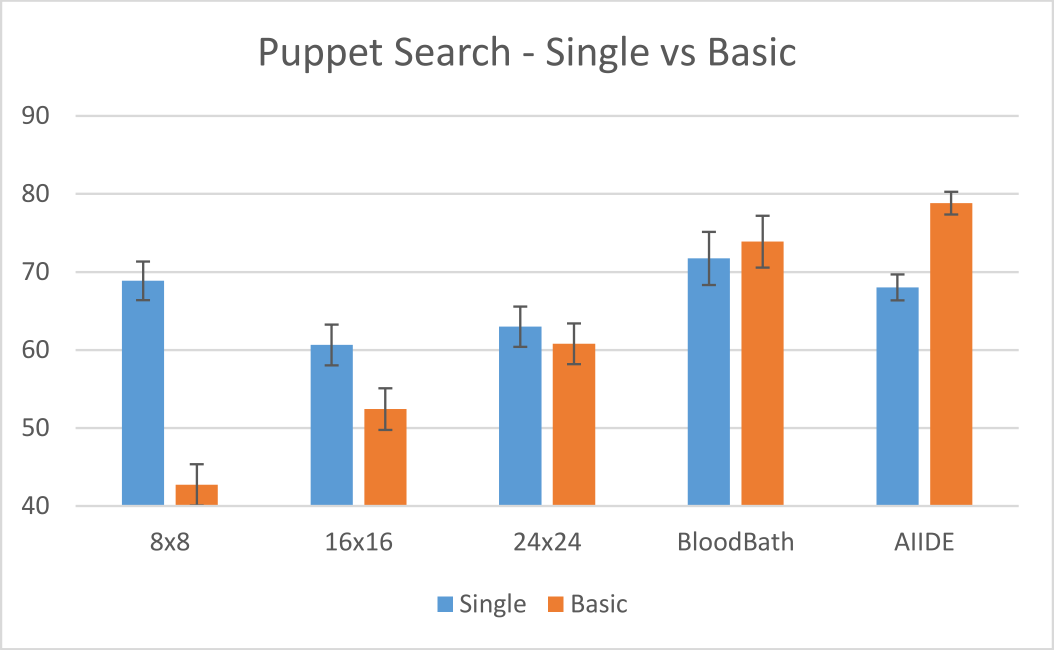


Figure 7.6: Average performance of *Puppet Search* versions using a single choice point or the basic script. Error bars show one standard error.

delayed. Therefore, deep search is important. In smaller maps, however, acting on recent game state information seems more important, as the actions' effects propagate faster. As expected, computing a plan every time an action needs

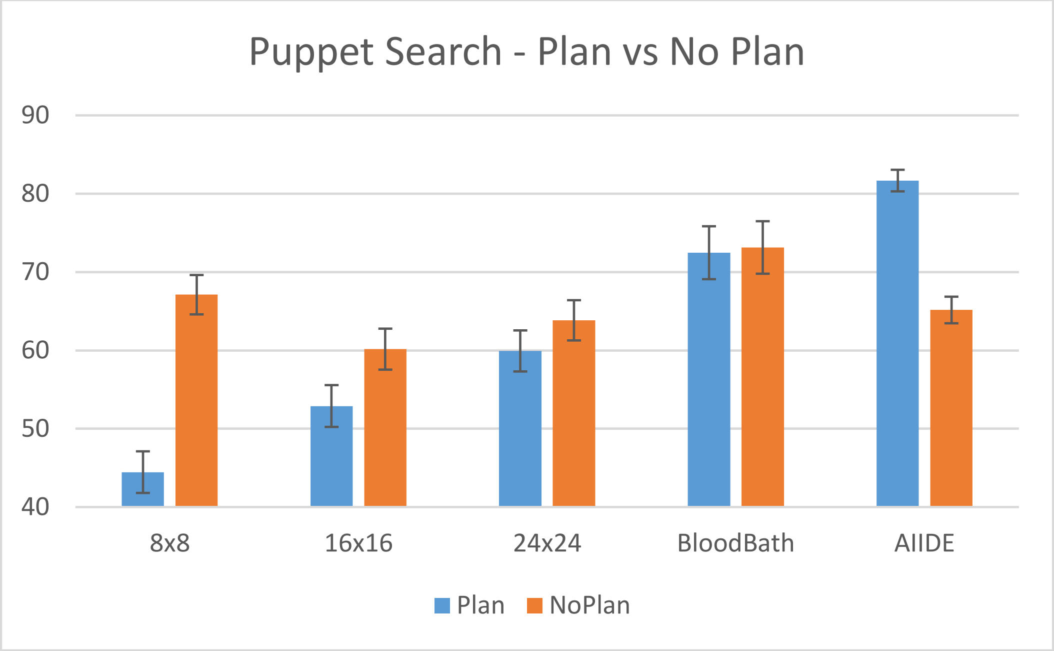


Figure 7.7: Average performance of *Puppet Search* using a standing plan or replanning every time. Error bars show one standard error.

to be issued instead of re-using a standing plan leads to better performance.

Table 7.1 shows win rates for all algorithms on all map sets. *PuppetABCDSingle* consistently outperforms its four constituent scripts in all but the smallest map, in which there is a clearly dominating strategy, and any deviation from it results in lower performance. After watching a few games, it seems clear that this demeanor — the total is more than the sum of its parts — is driven by some emergent patterns. Behaviors that were not included in the scripts begin to appear. Two in particular seem worthy of mention: *Puppet Search* will often start with a *WorkerRush* and later switch to training stronger units and use its previously built workers for gathering resources, thus bypassing the scripts' hard-coded decision to only use one gathering worker. Another similar behavior is that of switching back and forth between a script that trains ranged units and one that trains stronger melee units. The latter ones protect the weak ranged units, while both engage enemies simultaneously.

Table 7.1 also shows that NaïveMCTS cannot scale to the larger maps, and corroborates results in [65] that AHTN-F is stronger than AHTN-P in small maps, but it doesn't scale as well (though still better than NaïveMCTS). Worth noting is that two instances of *Puppet Search*— PuppetABCDSingle and PuppetUCTCDBasicNoPlan — can outperform almost all of the non *Puppet Search* agents in all maps, except for the *WorkerRush* on the 8x8 map.

Table 7.1: Average tournament win rates grouped by map type. 95% confidence intervals in parenthesis.

Player	8x8	16x16	24x24	BloodBath	AIIDE
WorkerRush	76.5(±4.5)	55.5(±5.3)	30.5(±4.9)	30.4(±6.8)	10.5(±1.9)
LightRush	24.0(±4.5)	46.1(±5.3)	66.2(±5.0)	33.0(±7.1)	46.7(±3.5)
RangedRush	12.6(±3.5)	12.2(±3.5)	23.2(±4.5)	15.5(±5.5)	28.8(±3.2)
HeavyRush	9.2(±3.0)	25.7(±4.6)	46.0(±5.3)	19.9(±6.0)	41.7(±3.4)
NaiveMCTS	53.7(±5.2)	57.4(±5.2)	33.9(±4.9)	3.0(±1.8)	8.2(±1.3)
AHTN-P	54.6(±5.2)	53.0(±5.3)	34.5(±5.1)	28.9(±6.7)	14.7(±2.3)
AHTN-F	73.1(±4.7)	47.8(±5.3)	20.5(±4.3)	36.9(±7.1)	12.0(±2.0)
PuppetUCTCDSingle	65.5(±5.0)	61.2(±5.2)	65.5(±5.1)	69.6(±7.0)	78.5(±2.9)
PuppetABCDSingle	68.5(±4.9)	66.1(±5.0)	68.8(±4.9)	79.5(±6.1)	83.0(±2.7)
PuppetUCTCDSingleNoPlan	70.7(±4.7)	66.5(±5.0)	53.1(±5.3)	62.5(±7.3)	44.9(±3.5)
PuppetABCDSingleNoPlan	70.8(±4.7)	48.8(±5.3)	64.6(±5.1)	75.3(±6.4)	65.7(±3.3)
PuppetUCTCDBasic	21.3(±4.4)	43.3(±5.3)	53.3(±5.3)	65.2(±7.2)	77.2(±3.0)
PuppetABCDBasic	22.6(±4.4)	41.1(±5.2)	52.2(±5.3)	75.6(±6.5)	88.0(±2.3)
PuppetUCTCDBasicNoPlan	63.1(±5.1)	65.5(±5.0)	71.6(±4.8)	69.3(±6.9)	68.3(±3.3)
PuppetABCDBasicNoPlan	63.8(±5.0)	59.8(±5.2)	66.1(±5.0)	85.4(±5.3)	81.8(±2.7)

7.4 Related Work

NaïveMCTS [62] uses smart action sampling for dealing with large action spaces. It treats the problem of assigning actions to individual units as a Combinatorial Multi-Armed Bandit (CMAB) problem, that is, a bandit problem with multiple variables. Each variable represents a unit, and the legal actions for each of those units are the values that each variable can take. Each variable is treated separately rather than translated to a regular Multi-Armed Bandit (MAB) (by considering that each possible legal value combination is a different arm) as in UCTCD. Samples are taken assuming that the reward of the combination of the arms is just the sum of the rewards for each arm (which they call the *naive assumption*). Each arm is treated as an independent local MAB and the individual samples are combined into a global MAB. The algorithm (naïveMCTS) was compared against other algorithms that sample or explore all possible low-level moves, such as ABCD and UCTCD. It outperformed them all, in three μRTS scenarios, with the biggest advantages found on the more complex scenarios.

Adversarial Hierarchical Task Networks (AHTNs) [65] are an alternative approach, that instead of sampling from the full action space, uses scripted actions to reduce the search space. It combines minimax tree search with HTN planning.

The authors implement five different AHTNs: 1. one with only the **Low Level** actions available in the game, which produces a game tree identical to one traversed by minimax search applied to raw low-level actions; 2. **Low Level** actions plus **Pathfinding**; 3. **Portfolio**, in which the main task of the game can be achieved only by three non-primitive tasks that encode three different hard-coded *rush* strategies, thus yielding a game tree with only one choice node at the top; 4. **Flexible**, with non-primitive tasks for harvesting resources, training units of different types, and attacking the enemy; and 5. **Flexible Single Target**, similar to Flexible, but encoded in such a way that all units that are sent to attack are sent to attack the same target, to reduce the branching factor. Experiments are presented in μRTS, a small scale RTS game designed for academic research, against four different scripts: a random script biased towards attacking, and the three scripts used by the Portfolio AHTN. The latter three AHTNs show good results, with some evidence that the Portfolio AHTN is the most likely to scale well to more complex games. The experiments presented in the previous section back this claim, though the performance couldn't match *Puppet Search*'s.

The AHTN approach, particularly the portfolio version, seems to have similar capabilities to *Puppet Search*. Both have the ability to encode strategies as high level tasks with options to be explored by the search procedure. However, *Puppet Search* is much simpler to implement, by having the ability to reuse scripts already present in the game. The full AHTN framework, including the tree search algorithm, is implemented in around 3400 lines of Java code, compared to around 1600 for *Puppet Search*. The AHTN definitions take 559 lines of LISP for AHTN-F and 833 for AHTN-P, while *Puppet Search*'s scripts took 66 lines of Java code for the single choice point one (plus 620 reused from the scripts already provided by μRTS) and 447 lines for the more complex one. The vast performance discrepancy with *PuppetABCDSingle* is due to a key difference between the algorithms. In AHTN, game tree nodes are the possible decompositions of the HTN plans, with leaves where no further decomposition can be performed. In the AHTN-P example, if both players have three choices in a single choice point, the tree has exactly 9 leaves. In *PuppetABCDSingle*, the choices are applied, the game forwarded (Algorithm 6.1, line 9), and then the choices will be explored again and the tree will continue to grow as long as there is time left.

Hierarchical Adversarial Search [84, 83] is an approach that uses several layers at different abstraction levels, each with a goal and an abstract view of the game state. The top layer of their three layer architecture chooses a set of objectives needed to win the game, the middle layer generates possible plans to accomplish those objectives, and the bottom layer evaluates the resulting plans and executes them at the individual unit level. For search purposes the game is advanced at the lowest level and the resulting states are abstracted back up the hierarchy. The algorithm was tested in SparCraft, a StarCraft simulator that only supports basic combat. The top level objectives, therefore, were restricted to destroying all opponent units while defending their own bases. Though this algorithm is general enough to encompass a full RTS game, only combat-related experiments were conducted.

An algorithm combining state and action abstractions for adversarial search in RTS games was proposed in [103, 104]. It constructs an abstract representation of the game state by decomposing the map into connected regions and grouping units into squads of a single unit type in each of the regions. Actions are restricted to squad movement to a neighboring region, attacking an enemy squad in the same region or staying idle. The approach only deals with movement and combat, but in their experiments it was added as a module to an existing StarCraft bot, so that it could play a full game. However, the only experiments presented were against the built-in AI, a much weaker opponent than current state-of-the-art bots.

Finally, the main difficulty with applying *PuppetSearch*, or any look-ahead search technique to a commercial RTS game, such as StarCraft: Brood War, is the lack of an accurate forward model or simulator. In contrast with board games, where the forward model is precisely detailed in the rules of the game, and everyone can build its own simulator to use for search purposes, commercial games are usually closed source, which means we don't have access to a simulator, nor can we easily reverse engineer it. An earlier version of *PuppetSearch* [5] has been used in StarCraft, with encouraging results despite to the poor accuracy of the simulator used. Experiments in the current paper were performed in μRTS to better evaluate *PuppetSearch*'s performance without external confounding elements.

7.5 Conclusions and Future Work

We have 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. The decision tree structure of the scripts ensures that only the choice combinations that make sense for a particular game state will be explored. This reduces the search effort considerably, and because scripts can play entire games, we can use the previous plan for as long as it takes to produce an updated one.

Our experiments show a similar performance to top scripted and search based agents in small maps, while vastly outperforming them on larger ones. Even a script with a single choice point to choose between different strategies can outperform the other players in most scenarios. Furthermore, on larger maps, *Puppet Search* benefits from the ability to use a standing plan to issue actions, while taking more time to calculate a new plan, resulting in even stronger performance.

From a design point of view *Puppet Search* allows game designers — by using scripts — to keep control over the range of behaviors the AI system can perform, while the adversarial look-ahead search enables it to better evaluate action outcomes, making it a stronger and more believable enemy. Based on promising experimental results on RTS games, we expect this new search framework to perform well in any game for which scripted AI systems can be built.

As for the general idea of *Puppet Search*, we believe it has great potential to improve decision quality in other complex domains as well in which expert knowledge in form of non-deterministic scripts is available.

In the future, we would like to extend this framework to tackle games with partial observability by using state and strategy inference similar to Bayesian models for opening prediction [93] and plan recognition [94], and particle models for state estimation [111].

7.6 Contributions Breakdown

The majority of the work in this chapter was performed by Nicolas A. Barriga. Marius Stanescu provided the evaluation function used by the algorithm and helped run experiments and write the published article. Michael Buro supervised the work.

Chapter 8

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

This chapter is joint work with Marius Stanescu and Michael Buro. It is accepted for presentation [7] at the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE), 2017.

Abstract

A commonly used technique for managing AI complexity in real-time strategy (RTS) games is to use action and/or state abstractions. High-level abstractions can often lead to good strategic decision making, but tactical decision quality may suffer due to lost details. A competing method is to sample the search space which often leads to good tactical performance in simple scenarios, but poor high-level planning.

We propose to use a deep convolutional neural network (CNN) to select among a limited set of abstract action choices, and to utilize the remaining computation time for game tree search to improve low level tactics. The CNN is trained by supervised learning on game states labelled by *Puppet Search*, a strategic search algorithm that uses action abstractions. The network is then used to select a script — an abstract action — to produce low level actions for all units. Subsequently, the game tree search algorithm improves the tactical actions of a subset of units using a limited view of the game state only considering units close to opponent units.

Experiments in the μRTS game show that the combined algorithm results in higher win-rates than either of its two independent components and other state-of-the-art μRTS agents.

To the best of our knowledge, this is the first successful application of a 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.

8.1 Introduction

In recent years, numerous challenging research problems have attracted AI researchers to using real-time strategy (RTS) games as test-bed in several areas, such as case-based reasoning and planning [66], evolutionary computation [3], machine learning [95], deep learning [106, 36, 70] and heuristic and adversarial search [22, 5]. Functioning AI solutions to most RTS sub-problems exist, but combining those doesn't come close to human level performance[1].

To cope with large state spaces and branching factors in RTS games, recent work focuses on smart sampling of the search space [25, 64, 63, 65] and state and action abstractions [103, 83, 8]. The first approach produces strong agents for small scenarios. The latter techniques work well on larger problems because of their ability to make good strategic choices. However, they have limited tactical ability, due to their necessarily coarse-grained abstractions. One compromise would be to allocate computational time for search-based approaches to improve the tactical decisions, but this allocation would come at the expense of allocating less time to strategic choices.

We propose to train a deep convolutional neural network (CNN) to predict the output of *Puppet Search*, thus leaving most of the time free for use by a tactical search algorithm. *Puppet Search* is a strategic search algorithm that uses action abstractions and has shown good results, particularly in large scenarios. We will base our network on previous work on CNNs for state evaluation [87], reformulating the earlier approach to handle larger maps.

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.

The remainder of this paper is organized as follows: Section 8.2 discussed previous related work, Section 8.3 describes our proposed approach and Section 8.4 provides experimental results. We then conclude and outline future work.

8.2 Related Work

Ever since the revolutionary results in the *ImageNet* competition [55], CNNs have been applied successfully in a wide range of domains. Their ability to learn hierarchical structures of spatially invariant local features make them ideal in settings that can be represented spatially. These include uni-dimensional streams in natural language processing [28], two-dimensional board games [81], or three-dimensional video analysis [48].

[1] `http://www.cs.mun.ca/~dchurchill/starcraftaicomp/report2015.shtml#mvm`

These diverse successes have inspired the application of CNNs to games. They have achieved human-level performance in several *Atari* games, by using Q-learning, a well known reinforcement learning (RL) algorithm [61]. But the most remarkable accomplishment may be AlphaGo [81], a *Go* playing program that last year defeated Lee Sedol, one of the top human professionals, a feat that was thought to be at least a decade away. As much an engineering as a scientific accomplishment, it was achieved using a combination of tree search and a series of neural networks trained on millions of human games and self-play, running on thousands of CPUs and hundreds of GPUs.

These results have sparked interest in applying deep learning to games with larger state and action spaces. Some limited success has been found in micromanagement tasks for RTS games [106], where a deep network managed to slightly outperform a set of baseline heuristics. Additional encouraging results were achieved for the task of evaluating RTS game states [87]. The network significantly outperforms other state-of-the-art approaches at predicting game outcomes. When it is used in adversarial search algorithms, they perform significantly better than using simpler evaluation functions that are three to four orders of magnitude faster.

Most of the described research on deep learning in multi-agent domains assumes full visibility of the environment and lacks communication between agents. Recent work addresses this problem by learning communication between agents alongside their policy [89]. In their model, each agent is controlled by a deep network which has access to a communication channel through which they receive the summed transmissions of other agents. The resulting model outperforms models without communication, fully-connected models, and models using discrete communication on simple imperfect information combat tasks. However, symmetric communication prevents handling heterogeneous agent types, limitation later removed by [70] which use a dedicated bi-direction communication channel and recurrent neural networks. This would be an alternative to the search algorithm we use for the tactical module on section 8.4.3, in cases where there is no forward model of the game, or there is imperfect information.

A new search algorithm that has shown good results particularly in large RTS scenarios, is *Puppet Search* [5, 6, 8]. It is an action abstraction mechanism that uses fast scripts with a few carefully selected choice points. These scripts are usually hard-coded strategies, and the number of choice points will depend on the time constraints the system has to meet. These choice points are then exposed to an adversarial look-ahead procedure, such as Alpha-Beta or Monte Carlo Tree Search (MCTS). The algorithm then uses a forward model of the game to examine the outcome of different choice combinations and decide on the best course of action. Using a restricted set of high-level actions results in low branching factor, enabling deep look-ahead and favouring strong strategic decisions. Its main weakness is its rigid scripted tactical micromanagement, which led to modest results on small sized scenarios where good micromanagement is key to victory.

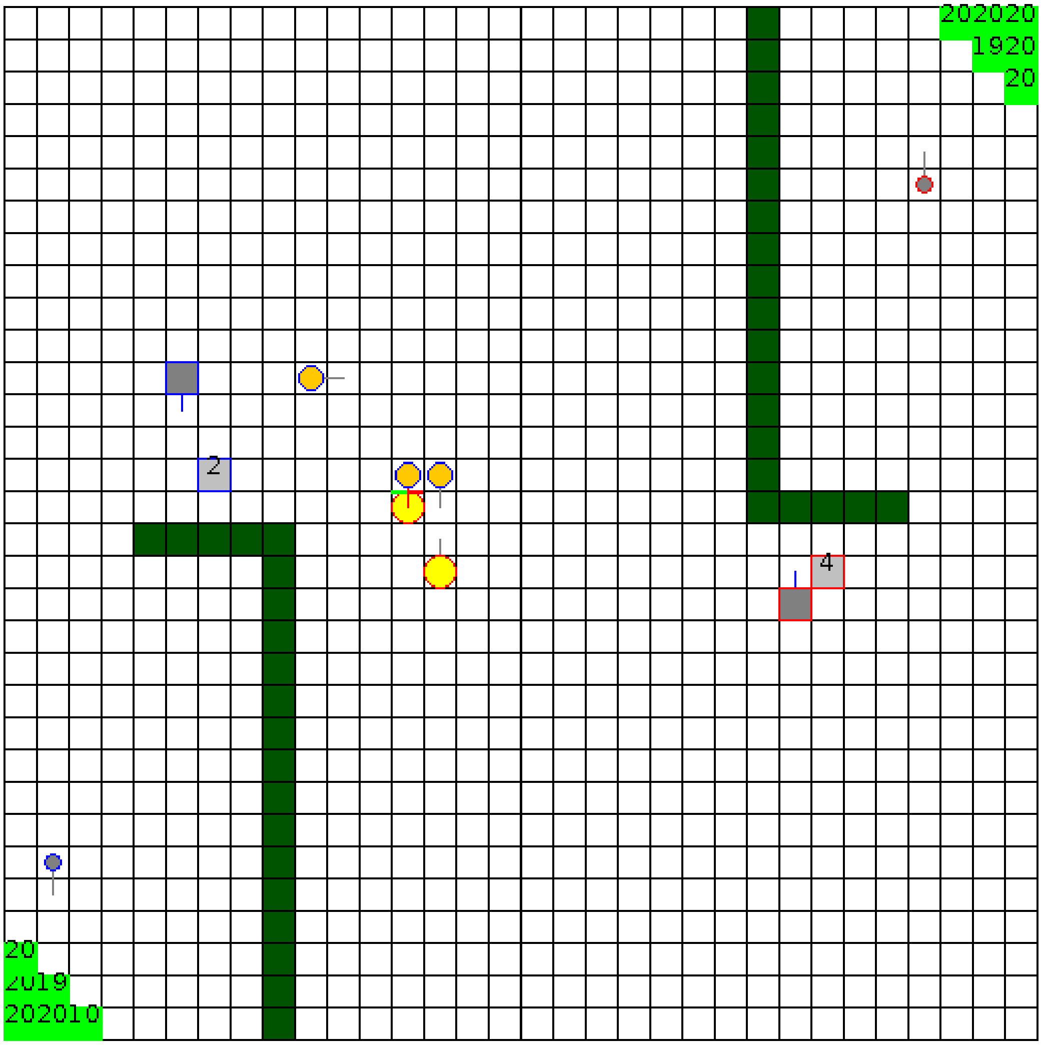


Figure 8.1: μRTS screenshot from a match between scripted LightRush and HeavyRush agents. Light green squares are resources, dark green are walls, dark grey are barracks and light grey the bases. Numbers indicate resources. Grey circles are worker units, small yellow circles are light combat units and big yellow ones are heavy combat units. Blue lines show production, red lines an attack and grey lines moving direction. Units are outlined blue (player 1) and red (player 2). μRTS can be found at `https://github.com/santiontanon/microrts`.

8.3 Algorithm Details

We build on previous work on RTS game state evaluation [87] applied to μRTS (see figure 8.1). This study presented a neural network architecture and experiments comparing it to simpler but faster evaluation functions. The

Table 8.1: Input feature planes for Neural Network. 25 planes for the evaluation network and 26 for the policy network.

Feature	# of planes	Description
Unit type	6	Base, Barracks, worker, light, ranged, heavy
Unit health points	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
Player	1	Player for which to select strategy

CNN-based evaluation showed a higher accuracy at evaluating game states. In addition, when used by state-of-the-art search algorithms, they perform significantly better than the faster evaluations. Table 8.1 lists the input features their network uses.

The network itself is 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 only consists of intermediate convolutional layers [82] and has the advantage of being an architecture that can fit a wide range of board sizes.

Table 8.2 shows the architectures of the evaluation network and the policy network we use, which only differ in the first and last layers. The first layer of the policy network has an extra plane which indicates which player's policy it is computing. The last layer of the evaluation network has two outputs, indicating if the state is a player 1 or player 2 win, while the policy network has four outputs, each corresponding to one of four possible actions. The global averaging used after the convolutional layers does not use any extra weights, compared to a fully connected layer. The benefit is that the number of network parameters does not grow when the map size is increased. This allows for a network to be quickly pre-trained on smaller maps, and then fine-tuned on the larger target map.

Puppet Search requires a forward model to examine the outcome of different actions and then choose the best one. Most RTS games do not have a dedicated forward model or simulator other than the game itself. This is usually too slow to be used in a search algorithm, or even unavailable due to technical constraints such as closed source code or being tied to the graphics engine. Using a policy network for script selection during game play allows us to bypass the need for a forward model of the game. Granted, the forward model

Table 8.2: Neural Network Architecture

<table>
<tr><th>Evaluation Network</th><th>Policy Network</th></tr>
<tr><td>Input 128x128, 25 planes</td><td>Input 128x128, 26 planes</td></tr>
<tr><td colspan="2">2x2 conv. 32 filters, pad 1, stride 1, LReLU
Dropout 0.2</td></tr>
<tr><td colspan="2">3x3 conv. 32 filters, pad 0, stride 2, LReLU
Dropout 0.2</td></tr>
<tr><td colspan="2">2x2 conv. 48 filters, pad 1, stride 1, LReLU
Dropout 0.2</td></tr>
<tr><td colspan="2">3x3 conv. 48 filters, pad 0, stride 2, LReLU
Dropout 0.2</td></tr>
<tr><td colspan="2">2x2 conv. 64 filters, pad 1, stride 1, LReLU
Dropout 0.2</td></tr>
<tr><td colspan="2">3x3 conv. 64 filters, pad 0, stride 2, LReLU
Dropout 0.2</td></tr>
<tr><td colspan="2">1x1 conv. 64 filters, pad 0, stride 1, LReLU</td></tr>
<tr><td>1x1 conv. 2 filters
pad 0, stride 1, LReLU</td><td>1x1 conv. 4 filters
pad 0, stride 1, LReLU</td></tr>
<tr><td colspan="2">Global averaging over 16x16 planes</td></tr>
<tr><td>2-way softmax</td><td>4-way softmax</td></tr>
</table>

is still required during the supervised training phase, but execution speed is less of an issue in this case, because training is performed offline. Training the network via reinforcement learning would remove this constraint completely.

Finally, with the policy network running significantly faster (3ms versus a time budget of 100ms per frame for search-based agents) than *Puppet Search* we can use the unused time to refine tactics. While the scripts used by *Puppet Search* and the policy network represent different strategic choices, they all share very similar tactical behaviour. Their tactical ability is weak in comparison to state-of-the-art search-based bots, as previous results [8] suggest.

For this reason, the proposed algorithm combines an FCN for strategic decisions and an adversarial search algorithm for tactics. The strategic component handles macro-management: unit production, workers, and sending combat units towards the opponent. The tactical component handles micromanagement during combat.

The complete procedure is described by Algorithm 8.1. It first builds a limited view of the game state, which only includes units that are close to opposing units (line 2). If this limited state is empty, all available computation time is assigned to the strategic algorithm, otherwise, both algorithms receive a fraction of the total time available. This fraction is decided empirically for each particular algorithm combination. Then, in line 9 the strategic algorithm is used to compute actions for all units in the state, followed by the tactical algorithm that computes actions for units in the limited state. Finally, the

Algorithm 8.1. Combined Strategy and Tactics

```
 1: procedure GETCOMBINEDACTION($(state, stratAI,$
                                 $tactAI,$
                                 $stratTime,$
                                 $tactTime)$
 2:     $limState \leftarrow$ EXTRACTCOMBAT($state$)
 3:     if ISEMPTY($limState$) then
 4:         SETTIME($stratAI, stratTime + tactTime$)
 5:     else
 6:         SETTIME($stratAI, stratTime$)
 7:         SETTIME($tactAI, tactTime$)
 8:     end if
 9:     $stratActions \leftarrow$ GETACTION($stratAI, state$)
10:     $tactActions \leftarrow$ GETACTION($tactAI, limState$)
11:     return MERGE($stratActions, tactActions$)
12: end procedure
```

actions are merged (line 11) by replacing the strategic action in case both algorithms produced actions for a particular unit.

8.4 Experiments and Results

All experiments were performed in machines running Fedora 25, with an Intel Core i7-7700K CPU, with 32GB of RAM and an NVIDIA GeForce GTX 1070 with 8GB of RAM. The Java version used for μRTS was OpenJDK 1.8.0, Caffe git commit 365ac88 was compiled with g++ 5.3.0, and pycaffe was run using python 2.7.13.

The *Puppet Search* version we used for all the following experiments utilizes alpha-beta search over a single choice point with four options. The four options are *WorkerRush*, *LightRush*, *RangedRush* and *HeavyRush*, and were also used as baselines in the following experiments. More details about these scripts can be found in [87].

Two other recent algorithms were also used as benchmarks, Na¨veMCTS [62] and Adversarial Hierarchical Task Networks (AHTNs) [65]. Na¨veMCTS is an MCTS variant with a sampling strategy that exploits the tree structure of Combinatorial Multi-Armed Bandits — bandit problems with multiple variables. Applied to RTS games, each variable represents a unit, and the legal actions for each of those units are the values that each variable can take. NaïveMCTS outperforms other game tree search algorithms on small scenarios. AHTNs are an alternative approach, similar to *Puppet Search*, that instead of sampling from the full action space, uses scripted actions to reduce the search space. It combines minimax tree search with HTN planning.

All experiments were performed on 128x128 maps ported from the *Star-*

Craft: Brood War maps used for the AIIDE competition. These maps, as well as implementations of *Puppet Search*, the four scripts, AHTN and NaïveMCTS are readily available in the μRTS repository.

8.4.1 State Evaluation Network

The data for training the evaluation network was generated by running games between a set of bots using 5 different maps, each with 12 different starting positions. Ties were discarded, and the remaining games were split into 2190 training games, and 262 test games. 12 game states were randomly sampled from each game, for a total of 26,280 training samples and 3,144 test samples. Data is labelled by a Boolean value indicating whether the first player won. All evaluation functions were trained on the same dataset.

The network's weights are initialized using Xavier initialization [40]. We used adaptive moment estimation (ADAM) [50] with default values of $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$ and a base learning rate of 10^{-4}. The batch size was 256.

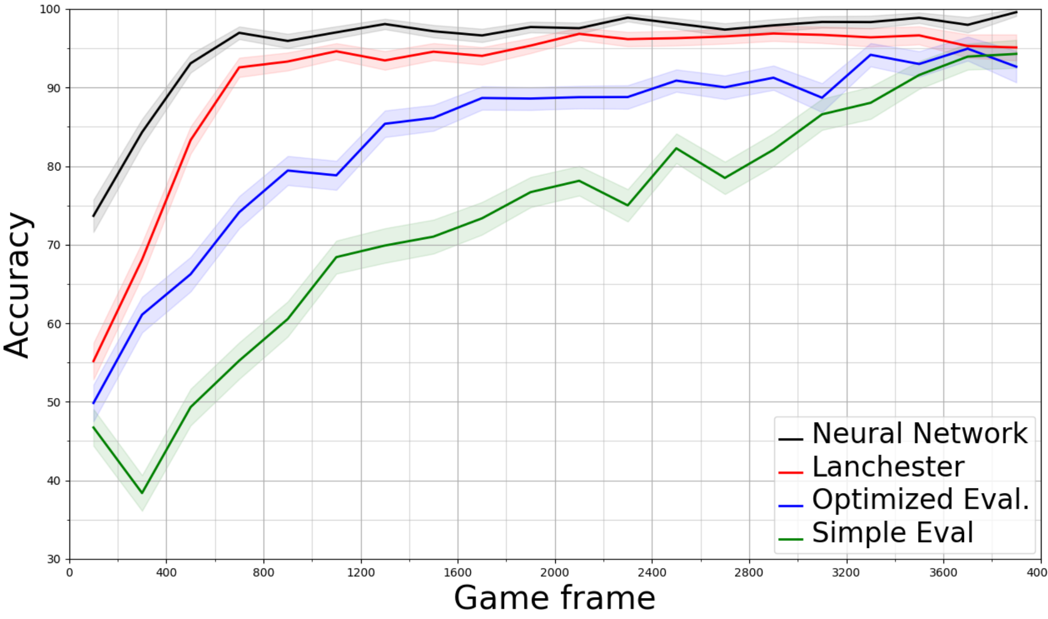


Figure 8.2: Comparison of evaluation accuracy between the neural network and the built-in evaluation function in μRTS. The accuracy of predicting the game winner is plotted against game time. Results are aggregated in 200-frame buckets. Shaded areas represent one standard error.

The evaluation network reaches 95% accuracy in classifying samples as wins or losses. Figure 8.2 shows the accuracy of different evaluation functions as game time progresses. The functions compared are the evaluation network, Lanchester [85], the simple linear evaluation with hard-coded weights that comes with μRTS, and a version of the simple evaluation with weights optimized using logistic regression. The network's accuracy is even higher than previous results in 8x8 maps [87]. The accuracy drop of the simple evaluation in the early game happens because it does not take into account units currently being built. If a player invests resources in units or buildings that take a long time to complete, its score lowers, despite the stronger resulting position after their completion. The other functions learn appropriate weights to mitigate this issue.

Table 8.3 shows the performance of *PuppetSearch* when using the Lanchester evaluation function and the neural network. The performance of the network is significantly better (P-value = 0.0011) than Lanchester's, even though the network is three orders of magnitude slower. Evaluating a game state using Lanchester takes an average of 2.7μs, while the evaluation network uses 2,574μs.

Table 8.3: Evaluation network versus Lanchester: round-robin tournament using 60 different starting positions per match-up and **100ms of computation time.**

	PS CNN	PS Lanc.	Light Rush	Heavy Rush	Avg.
PS CNN	-	59.2	89.2	72.5	73.6
PS Lanc.	40.8	-	64.2	67.5	57.5
LightRush	10.8	35.8	-	71.7	39.4
HeavyRush	27.5	32.5	28.3	-	29.4

Table 8.4: Evaluation network versus Lanchester: round-robin tournament on 20 different starting positions per match-up, searching to **depth 4.**

	PS CNN	PS Lanc.	Light Rush	Heavy Rush	Avg.
PS CNN	-	80	95	82.5	85.8
PS Lanc.	20	-	82.5	90	64.2
LightRush	5	17.5	-	70	30.8
HeavyRush	17.5	10	30	-	19.2

Table 8.4 shows the same comparison, but with *Puppet Search* searching to a fixed depth of 4, rather than having 100ms per frame. The advantage of the neural network is much more clear, as execution speed does not matter in this case. (P-value = 0.0044)

8.4.2 Policy Network

We used the same procedure as in the previous subsection, but now we labelled the samples with the outcome of a 10 second *Puppet Search* using the evaluation network. The resulting policy network has an accuracy for predicting the

Table 8.5: Policy network versus Puppet Search: round-robin tournament using 60 different starting positions per match-up.

	PS	Policy Net.	Light Rush	Heavy Rush	Ranged Rush	Worker Rush	Avg.
PS	-	55.8	87.5	66.67	91.7	93.3	65.8
Policy net.	44.2	-	94.2	71.7	100	61.7	61.9
LightRush	12.5	5.8	-	71.7	100	100	48.3
HeavyRush	33.3	28.3	28.3	-	100	100	48.3
RangedRush	8.3	0	0	0	-	100	18.1
WorkerRush	6.7	38.3	0	0	0	-	7.5

correct puppet move of 73%, and a 95% accuracy for predicting any of the top 2 moves.

Table 8.5 shows the policy network coming close to *Puppet Search* and defeating all the scripts.

8.4.3 Strategy and Tactics

Finally, we compare the performance of the policy network and *Puppet Search* as the strategic part of a combined strategic/tactical agent. We will do so by assigning a fraction of the allotted time to the strategic algorithm and the remainder to the tactical algorithm, which will be NaïveMCTS in our experiments. We expect the policy network to perform better in this scenario, as it runs significantly faster than *Puppet Search* while maintaining similar action performance.

The best time split between strategic and tactical algorithm was determined experimentally to be 20% for *Puppet Search* and 80% for NaïveMCTS. The policy network uses a fixed time (around 3ms), and the remaining time is assigned to the tactical search.

Table 8.6 shows that both strategic algorithms greatly benefit from blending with a tactical algorithm. The gains are more substantial for the policy network, which now scores 56.7% against its *Puppet Search* counterpart. It also has a 4.3% higher overall win rate despite markedly poorer results against WorkerRush and AHTN-P. These seems to be due to a strategic mistake on the part of the policy network, which, if its cause can be detected and corrected, would lead to even higher performance.

8.5 Conclusions and Future Work

We have extended previous research that used CNNs to accurately evaluate RTS game states in small maps to larger map sizes usually used in commercial

Table 8.6: Mixed Strategy/Tactics agents: round-robin tournament using 60 different starting positions per match-up.

	Policy Naïve	PS Naïve	PS	Policy Network	Light Rush	Heavy Rush	Ranged Rush	AHTN P	Worker Rush	Naïve MCTS	Avg.
Policy net.-Naïve	-	56.7	97.5	100.0	100.0	95.8	100.0	72.5	74.2	98.3	88.3
PS-Naïve	43.3	-	81.7	79.2	90.0	94.2	93.3	90.0	90.8	93.3	84.0
PS	2.5	18.3	-	63.3	86.7	69.2	92.5	96.7	95.0	93.3	68.6
Policy net.	0.0	20.8	36.7	-	94.2	71.7	100.0	57.5	61.7	97.5	60.0
LightRush	0.0	10.0	13.3	5.8	-	71.7	100.0	100.0	100.0	96.7	55.3
HeavyRush	4.2	5.8	30.8	28.3	28.3	-	100.0	100.0	100.0	74.2	52.4
RangedRush	0.0	6.7	7.5	0.0	0.0	0.0	-	100.0	100.0	86.7	33.4
AHTN-P	27.5	10.0	3.3	42.5	0.0	0.0	0.0	-	64.2	68.3	24.0
WorkerRush	25.8	9.2	5.0	38.3	0.0	0.0	0.0	35.8	-	71.7	20.6
NaïveMCTS	1.7	6.7	6.7	2.5	3.3	25.8	13.3	31.7	28.3	-	13.3

RTS games. The average win prediction accuracy at all game times is higher compared to smaller scenarios. This is probably the case because strategic decisions are more important than tactical decisions in larger maps, and strategic development is easier to quantify by the network. Although the network is several orders of magnitude slower than competing simpler evaluation functions, its accuracy makes it more effective. When the *Puppet Search* high-level adversarial search algorithm uses the CNN, its performance is better than when using simpler but faster functions.

We also trained a policy network to predict the outcome of *Puppet Search*. The win rate of the resulting network is similar to that of the original search, with some exceptions against specific opponents. However, while slightly weaker in playing strength, a feed-forward network pass is much faster. This speed increase created the opportunity for using the saved time to fix the shortcomings introduced by high-level abstractions. A tactical search algorithm can micro-manage units in contact with the enemy, while the policy chosen by the network handles routine tasks (mining, marching units toward the opponent) and strategic tasks (training new units). The resulting agent was shown to be stronger than the policy network alone in all tested scenarios, but can only partially compensate for the network's weaknesses against specific opponents.

Looking into the future, we recognize that most tactical search algorithms, like the MCTS variant we used, have the drawback of requiring a forward model of the game. Using machine learning techniques to make tactical decisions would eliminate this requirement. However, this has proved to be a difficult goal, as previous attempts by other researchers have had limited success on simple scenarios [106, 98]. Recent research avenues based on integrating concepts such as communication [89], unit grouping and bidirectional recurrent neural networks [70] suggest that strong tactical networks might soon be available.

The network architecture presented in this paper, being fully convolutional, can be used on maps of any (reasonable) size without increasing its number of parameters. Hence, future research could include assessing the speed-up obtained by taking advantage of *transfer learning* from smaller maps to larger ones. Also of interest would be to determine whether different map sizes can be mixed within a training set. It would also be interesting to investigate the performance of the networks on maps that have not previously been seen during training.

Because the policy network exhibits some weaknesses against specific opponents, further experiments should be performed to establish whether this is due to a lack of appropriate game state samples in the training data or other reasons. A related issue is our reliance on labelled training data, which could be resolved by using reinforcement learning techniques, such as DQN (deep Q network) learning. However, full RTS games are difficult for these techniques, mainly because the only available reward is the outcome of the game. In addition, action choices near the endgame (close to the reward), have very little impact on the outcome of the game, while early ones (when there is no reward), matter most. There are several strategies available that could help overcome these issues, such as curriculum learning [9], reward shaping [32], or implementing double DQN learning [44]. These strategies have proved useful on adversarial games, games with sparse rewards, or temporally extended planning problems respectively.

8.6 Contributions Breakdown and Updates Since Publication

Most of the work in this chapter was performed by Nicolas A. Barriga. Marius Stanescu helped with the design of the evaluation network and provided Caffe scripts. 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 [44] with experience replay [61] and used the value network to provide a virtual reward. In all of our experiments the network converges to always selecting the script with the best average performance, regardless of the game state.

We believe there are two issues that complicate learning. First of all, the rewards are very sparse as they are only generated when a match ends. Secondly, the choice of actions near the end of the game is mostly inconsequential, because any action (script choice) will be 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. Moreover, when action choices matter most, at the beginning of the game, the virtual rewards generated by the value network are very small and noisy. We attempted to overcome this complication by starting the learning process on using endgame scenarios and slowly adding earlier states until full games were played, but without success.

Chapter 9

Conclusions and Future Work

In this chapter we summarize the contributions in this dissertation and suggest future applications and research avenues.

9.1 Conclusions

Chapter 3 presented *Multiblock Parallel* UCT, an MCTS variant that takes advantage of GPU hardware resources to speedup computations. The strong outcome in terms of speed and playing strength in the game of Ataxx led to this work's nomination for a best paper award at the CIG 2014 conference. However, the results are nowhere near what would be needed to tackle the large branching factors and tight time constraints of RTS games.

Instead of the brute-force approach of the previous chapter, chapter 4 performs a hierarchical decomposition of the search space. Several layers are given a partial or abstract view of the game state, and a goal to accomplish, in order to reduce the overall branching factor of the search. Despite promising results in limited scenarios, we discontinued this approach due to the difficulties in hand-crafting these ad-hoc layers.

Chapter 5 presented a genetic algorithm to solve the building placement problem in RTS games. This is an example of an RTS sub-problem solution that can be used as a black box in a script. Another example is path-finding, used by *Puppet Search* μRTS scripts in chapter 7, or build-order search [23] performed by StarCraft scripts in chapter 6.

Scripted agents incorporating this type of sub-problem solutions have shown to be very strong. Most top Starcraft: Brood War bots are built in this way. By designing a search framework to select among these scripts, or add flexibility to them, we can only make them stronger.

Chapters 6 and 7 present *Puppet Search*, an adversarial look-ahead search algorithm that uses configurable scripts with choice points as an action abstraction mechanism. Chapter 6 shows how even a simple implementation to select among four scripts, using a simulator built on very coarse abstractions, can produce a very capable *StarCraft: Brood War* agent. Chapter 7 explores the performance of several variants of *Puppet Search* in μRTS. The Alpha-beta variant seems to have better performance than the MCTS one, particularly on larger maps, very likely due to the low branching factors involved (between 4 and 8). Two scripts were tested, with 4 and 8 possible choices. The one with the least choices performed better 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, otherwise they are just increasing the branching factor of the search tree without providing any benefit. Finally, a variant that runs a search every time an action is needed was compared to a variant that spreads the search over 50 frames, and then uses the principal variation from that search as a standing plan. This plan is then used to generate actions in the game until the next search episode is done. The standing plan variant, which allows for deeper search, proved very useful on bigger maps, where playouts are usually slower and action consequences are delayed.

Chapter 8 first extends previous work on CNN for state evaluation, and shows the approach scales to larger maps, and search algorithms benefit from its accuracy, despite its speed disadvantages over common alternatives. Afterwards, a policy CNN is trained to predict the output of *Puppet Search* (with the evaluation network), in order to both speed-up strategy selection, and to eliminate the need of a forward model during gameplay. Finally, the policy network is combined with NaïveMCTS, a tactical search algorithm, to produce a strong μRTS agent capable of defeating *Puppet Search*, the policy network, the tactical search, a combined *Puppet Search*/NaïveMCTS, all the *Puppet Search* scripts individually and an AHTN — another state-of-the-art search algorithm.

In summary, we have presented configurable scripts as an action abstraction mechanism, and provided empirical evidence of their usefulness in adversarial domains. These scripts can incorporate solutions to particular sub-problems in the area of application. Examples in RTS games include pathfinding, building placement and build-order search. The abstract action choices can be decided by a search procedure or using machine learning techniques.

9.2 Future Research

Demonstrating the generality of *Puppet Search* and the use of configurable scripts as action abstractions would be very valuable. As previously mentioned, there is nothing in these ideas that is particular to adversarial environments. However, applications on single agent domains like the ones mentioned in section 1.1 are needed to assess whether *Puppet Search* can match — or surpass — state-of-the-art planning algorithms. Additionally, applications to different game genres would further enlighten us on the strengths and weaknesses of the techniques presented here.

Going back to videogames, if we are to build a system capable of defeating top professional RTS players, the main avenues for future research are on handling partial observability and on removing the necessity of a forward model, at least during games.

Most commercial RTS games exhibit partially observable environments, by means of *fog-of-war*, which limits the player's visibility to locations close to his units. Unexplored locations are not visible, while previously explored places only show terrain features, but not possible units that might be there. Several solutions have been proposed to tackle this issue, such as Bayesian models for opening prediction [93] and plan recognition [94], and particle models for state estimation [111]. Future work on deep networks will possibly show that they are capable of performing under partial observability constraints as well. In all likelihood, some combination of the above will be needed for human-level performance.

Games such as StarCraft, don't provide access to the internal forward model for use by look-ahead search algorithms. And, even if they did, it would probably be too slow for online decision making. One alternative is to build a simulator, a time consuming solution that probably won't be accurate enough. Low-level actions produced by SparCraft (see appendix B.2) can't be used directly in StarCraft, and the simulation used in chapter 6 was not precise enough to show the full strength of *Puppet Search*. A second option is to learn a forward model, but we are not aware of any work on learning the full model for an RTS game. Limited models have been learned for sub-problems such as combat [105]. Finally, we can use machine learning — rather than look-ahead — to directly choose actions, as we did in chapter 8. We still used a forward model during training of the policy network, and for the tactical search algorithm. The network could be trained via reinforcement learning, taking advantage of recent developments of curriculum learning [9], reward shaping [32], or double DQN learning [44]. The tactical search can be replaced by a tactical network. These networks haven't reached a high playing strength, but active research is ongoing, with some very recent promising results based on integrating concepts such as communication [89], unit grouping and bidirectional recurrent neural networks [70].

Finally, further efforts encouraging the adoption, by the videogame industry, of the techniques presented in this dissertation should be a undertaken. We have already published a tutorial version of chapter 6 as chapter in a book directed at industry professionals [6], and we feel this approach should be continued with the latest advances in the field.

Bibliography

[1] Louis Victor Allis. *Searching for solutions in games and artificial intelligence*. PhD thesis, University of Limburg, Maastricht, 1994.

[2] Radha-Krishna Balla and Alan Fern. UCT for tactical assault planning in real-time strategy games. In *IJCAI*, pages 40–45, 2009.

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

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

[12] 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.

[13] Tristan Cazenave and Nicolas Jouandeau. On the parallelization of UCT. In *Proceedings of the Computer Games Workshop*, pages 93–101. Citeseer, 2007.

[14] Tristan Cazenave and Nicolas Jouandeau. A parallel Monte-Carlo tree search algorithm. *Computers and Games*, pages 72–80, 2008.

[15] 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.

[16] Michal Čertickỳ. Implementing a wall-in building placement in StarCraft with declarative programming. *arXiv preprint arXiv:1306.4460*, 2013.

[17] Guillaume Chaslot, Mark Winands, and H Jaap van Den Herik. Parallel Monte-Carlo tree search. *Computers and Games*, pages 60–71, 2008.

[18] Michael Chung, Michael Buro, and Jonathan Schaeffer. Monte Carlo planning in RTS games. In *IEEE Symposium on Computational Intelligence and Games (CIG)*, 2005.

[19] David Churchill. 2013 AIIDE StarCraft AI competition report. http://www.cs.ualberta.ca/ cdavid/starcraftaicomp/ report2013.shtml, October 2013.

[20] David Churchill. SparCraft: open source StarCraft combat simulation. http://code.google.com/p/sparcraft/, 2013.

[21] David Churchill. *Heuristic Search Techniques for Real-Time Strategy Games.* PhD thesis, University of Alberta, 2016.

[22] David Churchill and Michael Buro. Build order optimization in StarCraft. In *AI and Interactive Digital Entertainment Conference, AIIDE (AAAI)*, pages 14–19, 2011.

[23] David Churchill and Michael Buro. Build order optimization in StarCraft. *Proceedings of AIIDE*, pages 14–19, 2011.

[24] David Churchill and Michael Buro. Incorporating search algorithms into RTS game agents. In *AI and Interactive Digital Entertainment Conference, AIIDE (AAAI)*, 2012.

[25] 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.

[26] David Churchill, Michael Preuss, Florian Richoux, Gabriel Synnaeve, Alberto Uriarte, Santiago Ontañón, and Michal Čertickỳ. StarCraft bots and competitions. *Springer Encyclopedia of Computer Graphics and Games*, 2016.

[27] 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.

[28] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167. ACM, 2008.

[29] Rémi Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. *Computers and Games*, pages 72–83, 2007.

[30] Caio Freitas de Oliveira and Charles Andryê Galvão Madeira. Creating efficient walls using potential fields in real-time strategy games. In *Computational Intelligence and Games (CIG), 2015 IEEE Conference on*, pages 138–145. IEEE, 2015.

[31] 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.

[32] Sam Devlin, Daniel Kudenko, and Marek Grześ. An empirical study of potential-based reward shaping and advice in complex, multi-agent systems. *Advances in Complex Systems*, 14(02):251–278, 2011.

[33] Abdelrahman Elogeel, Andrey Kolobov, Matthew Alden, and Ankur Teredesai. Selecting robust strategies in RTS games via concurrent plan augmentation. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, pages 155–162. International Foundation for Autonomous Agents and Multiagent Systems, 2015.

[34] Markus Enzenberger and Martin Müller. A lock-free multithreaded Monte-Carlo tree search algorithm. *Advances in Computer Games*, pages 14–20, 2010.

[35] 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.

[36] 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.

[37] 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.

[38] Sylvain Gelly. *A contribution to reinforcement learning; application to computer-Go.* PhD thesis, Universite Paris-Sud, 2008.

[39] Sylvain Gelly, Jean-Baptiste Hoock, Arpad Rimmel, Olivier Teytaud, Yann Kalemkarian, et al. On the parallelization of Monte-Carlo planning. In *ICINCO*, 2008.

[40] 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.

[41] David Edward Goldberg. *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley Professional, 1989.

[42] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[43] James Hanna, Jerry Reaper, Tim Cox, and Martin Walter. Course of action simulation analysis. Technical report, SCIENCE APPLICATIONS INTERNATIONAL CORP (SAIC) BEAVERCREEK OHIO, 2005.

[44] 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.

[45] Adam Heinermann. Broodwar API. http://code.google.com/p/bwapi/, 2014.

[46] Ryan Houlette and Dan Fu. The ultimate guide to FSMs in games. *AI Game Programming Wisdom 2*, 2003.

[47] 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.

[48] Shuiwang Ji, Wei Xu, Ming Yang, and Kai Yu. 3d convolutional neural networks for human action recognition. *IEEE transactions on pattern analysis and machine intelligence*, 35(1):221–231, 2013.

[49] Hideki Kato and Ikuo Takeuchi. Parallel monte-carlo tree search with simulation servers. In *13th Game Programming Workshop (GPW-08)*, 2008.

[50] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[51] Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *Machine Learning: ECML 2006*, pages 282–293. Springer, 2006.

[52] 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.

[53] Alexander Kott, Larry Ground, Ray Budd, Lakshmi Rebbapragada, and John Langston. Toward practical knowledge-based tools for battle planning and scheduling. In *Proceedings of the 14th conference on Innovative applications of artificial intelligence-Volume 1*, pages 894–899. AAAI Press, 2002.

[54] Alexander Kovarsky and Michael Buro. Heuristic search applied to abstract combat games. *Advances in Artificial Intelligence*, pages 66–78, 2005.

[55] 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.

[56] Frederick William Lanchester. *Aircraft in warfare: The dawn of the fourth arm.* Constable limited, 1916.

[57] Victor W Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, et al. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. *ACM SIGARCH computer architecture news*, 38(3):451–460, 2010.

[58] Siming Liu, Sushil J Louis, and Monica Nicolescu. Using CIGAR for finding effective group behaviors in RTS games. In *2013 IEEE Conference on Computational Intelligence in Games (CIG)*, pages 1–8. IEEE, 2013.

[59] Kinshuk Mishra, Santiago Ontañón, and Ashwin Ram. Situation assessment for plan retrieval in real-time strategy games. In *ECCBR*, pages 355–369, 2008.

[60] Melanie Mitchell. *An introduction to genetic algorithms.* MIT press, 1998.

[61] 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.

[62] Santiago Ontañón. The combinatorial multi-armed bandit problem and its application to real-time strategy games. In *AIIDE*, 2013.

[63] 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.

[64] Santiago Ontañón. Combinatorial multi-armed bandits for real-time strategy games. *Journal of Artificial Intelligence Research*, 58:665–702, 2017.

[65] 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.

[66] 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.

[67] Santiago Ontañón, Kinshuk Mishra, Neha Sugandh, and Ashwin Ram. Learning from demonstration and case-based planning for real-time strategy games. In Bhanu Prasad, editor, *Soft Computing Applications in Industry*, volume 226 of *Studies in Fuzziness and Soft Computing*, pages 293–310. Springer Berlin / Heidelberg, 2008.

[68] Santiago Ontañón, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill, and Michael Preuss. RTS AI problems and techniques. *Springer Encyclopedia of Computer Graphics and Games*, 2015.

[69] 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.

[70] 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.

[71] 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.

[72] Robert Rasch, Alexander Kott, and Kenneth D Forbus. Incorporating ai into military decision making: an experiment. *IEEE Intelligent Systems*, 18(4):18–26, 2003.

[73] Florian Richoux, Alberto Uriarte, and Santiago Ontanón. Walling in strategy games via constraint optimization. In *Tenth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2014.

[74] Glen Robertson and Ian Watson. A review of real-time strategy game ai. *AI Magazine*, 35(4):75–204, 2014.

[75] Kamil Rocki and Reiji Suda. Parallel Monte Carlo tree search on GPU. In *Eleventh Scandinavian Conference on Artificial Intelligence: Scai 2011*, volume 227, page 80. IOS Press, Incorporated, 2011.

[76] 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.

[77] Frederik Schadd, Sander Bakkes, and Pieter Spronck. Opponent modeling in real-time strategy games. In *GAMEON*, pages 61–70, 2007.

[78] Jonathan Schaeffer, Neil Burch, Yngvi Bjrnsson, Akihiro Kishimoto, Martin Mller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *Science*, 317(5844):1518–1522, 2007.

[79] 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.

[80] Alexander Shleyfman, Antonín Komenda, and Carmel Domshlak. On combinatorial actions and cmabs with linear side information. In *ECAI*, pages 825–830, 2014.

[81] 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.

[82] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*, 2014.

[83] 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.

[84] 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.

[85] 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.

[86] 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.

[87] 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.

[88] 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.

[89] 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.

[90] Omar Syed. The arimaa challenge: From inception to completion. *ICGA Journal*, 38(1):3–11, 2015.

[91] Omar Syed and Aamir Syed. Arimaa-a new game designed to be difficult for computers. *ICGA JOURNAL*, 26(2):138–139, 2003.

[92] Gabriel Synnaeve. *Bayesian programming and learning for multi-player video games*. PhD thesis, Université de Grenoble, 2012.

[93] 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.

[94] Gabriel Synnaeve and Pierre Bessière. A Bayesian model for plan recognition in RTS games applied to StarCraft. *Proceedings of AIIDE*, pages 79–84, 2011.

[95] 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.

[96] 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.

[97] 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.

[98] 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.

[99] Fabien Teytaud and Olivier Teytaud. Creating an upper-confidence-tree program for havannah. In *Advances in Computer Games*, pages 65–74. Springer, 2010.

[100] Julian Togelius, Mike Preuss, Nicola Beume, Simon Wessing, Johan Hagelbäck, and Georgios N Yannakakis. Multiobjective exploration of the StarCraft map space. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, pages 265–272. IEEE, 2010.

[101] Alberto Uriarte. *Adversarial Search and Spatial Reasoning in Real Time Strategy Games*. PhD thesis, Drexel University, 2017.

[102] Alberto Uriarte and Santiago Ontañón. Kiting in RTS games using influence maps. In *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2012.

[103] 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.

[104] 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.

[105] Alberto Uriarte and Santiago Ontanón. Combat models for rts games. *IEEE Transactions on Computational Intelligence and AI in Games*, 2017.

[106] 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.

[107] 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.

[108] Matthew Wall. GAlib: A C++ library of genetic algorithm components. http://lancet.mit.edu/ga/GAlib.html, 2007.

[109] 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.

[110] Ben G. Weber and Michael Mateas. A data mining approach to strategy prediction. In *IEEE Symposium on Computational Intelligence and Games (CIG)*, 2009.

[111] 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.

[112] Nick Wedd. Human-computer go challenges. http://www.computer-go.info/h-c/index.html#2013, April 2013.

[113] Stefan Wender and Ian Watson. Applying reinforcement learning to small scale combat in the real-time strategy game StarCraft:Broodwar. In *CIG (IEEE)*, 2012.

[114] 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.

[115] Kazuki Yoshizoe, Akihiro Kishimoto, Tomoyuki Kaneko, Haruhiro Yoshimoto, and Yutaka Ishikawa. Scalable distributed Monte-Carlo tree search. In *Fourth Annual Symposium on Combinatorial Search*, 2011.

[116] Albert L Zobrist. A new hashing method with application for game playing. *ICCA journal*, 13(2):69–73, 1970.

Appendix A

Other Research

This appendix contains the abstracts of further work I executed as part of my doctoral research. It is not included in the main body of this dissertation because my contributions were relatively minor. A tutorial version of the work described in section A.1 has also been published as a chapter in Game AI Pro 3, a book targeted at industry professionals [86].

A.1 Using Lanchester Attrition Laws for Combat Prediction in StarCraft

This section is the abstract of work led by Marius Stanescu. I mainly contributed by setting up experiments. Michael Buro supervised the work. The original full length version was published [85] at the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE), 2015.

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 modelling 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.

A.2 Evaluating Real-Time Strategy Game States Using Convolutional Neural Networks

This section is abstract of work led by Marius Stanescu. My main contribution was with μRTS expertise and experiments. Michael Buro supervised the work. ©2014 IEEE. Reprinted, with permission, from Marius Stanescu, Nicolas A. Barriga and Michael Buro, Evaluating Real-Time Strategy Game States Using Convolutional Neural Networks [87], IEEE Conference on Computational Intelligence and Games, 2016.

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 CNNs 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, 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.

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.

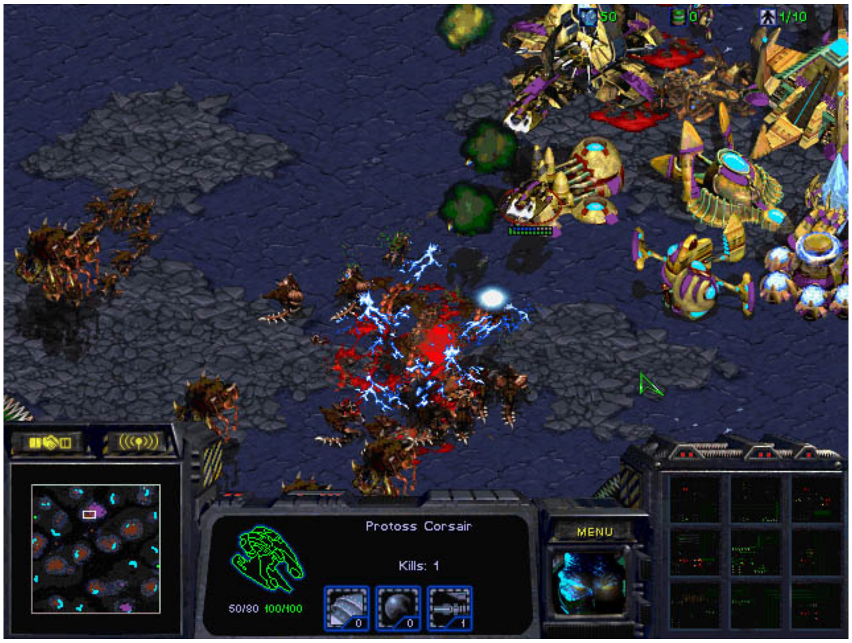


Figure B.1: Screenshot of StarCraft:Brood War, showing a group of Zerg assaulting a Protoss base.

Some characteristics of StarCraft are:

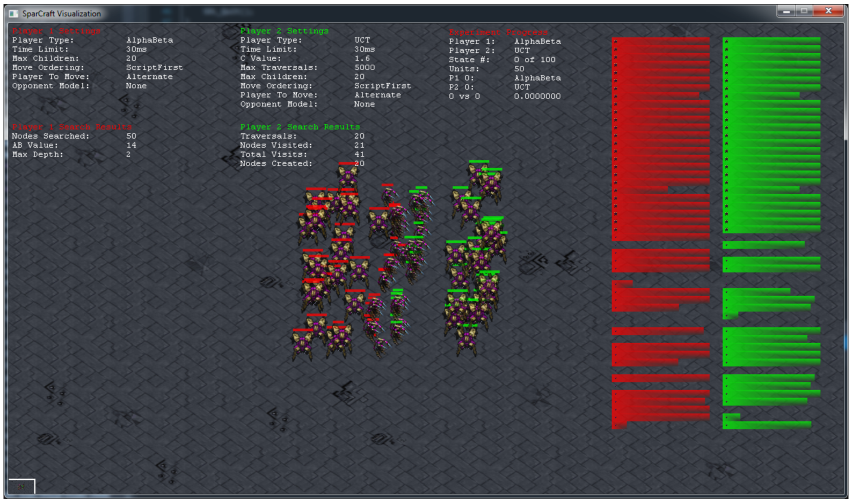
Real-time: The game runs at 24 simulation frames per second, and it moves on even if a player doesn't execute any actions.

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.



```
Player 1 Settings
Player Type:          AlphaBeta
Time Limit:           30ms
Max Children:         20
Move Ordering:        ScriptFirst
Player To Move:       Alternate
Opponent Model:       None

Player 1 Search Results
Nodes Searched:       50
AB Value:             14
Max Depth:            2
```

```
Player 2 Settings
Player Type:          UCT
Time Limit:           30ms
C Value:              1.6
Max Traversals:       5000
Max Children:         20
Move Ordering:        ScriptFirst
Player To Move:       Alternate
Opponent Model:       None

Player 2 Search Results
Traversals:           20
Nodes Visited:        21
Total Visits:         41
Nodes Created:        20
```

```
Experiment Progress
Player 1:    AlphaBeta
Player 2:    UCT
State #:     0 of 100
Units:       50
P1 0:        AlphaBeta
P2 0:        UCT
0 vs 0       0.0000000
```

Figure B.2: Screenshot of SparCraft, showing a combat experiment between Alpha-Beta and UCT based players.

State/Action Space: [69] 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

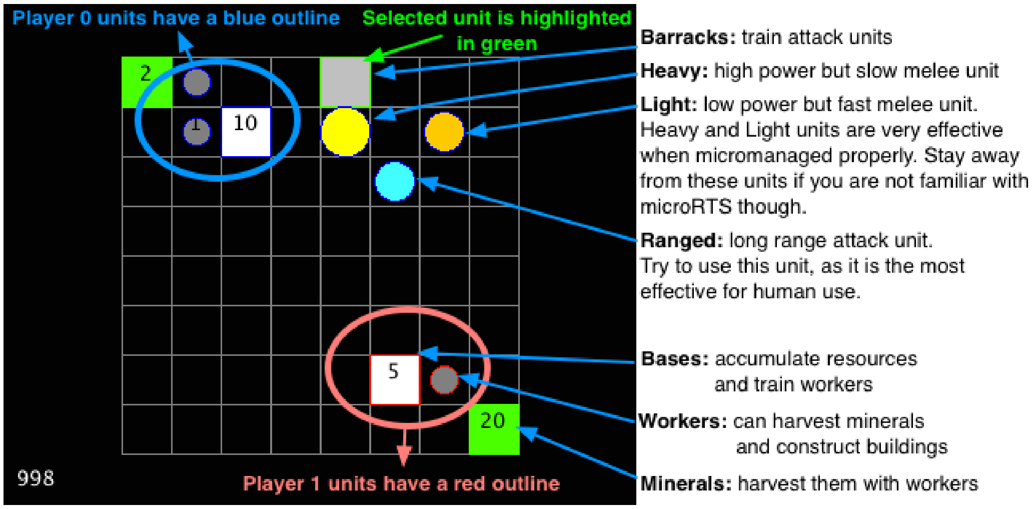
SparCraft is a StarCraft combat simulator written by David Churchill[1]. Its main design goal was to estimate the outcomes of battles as quickly as possible. To accomplish this it greatly simplifies the game. There are no collisions, thus, pathfinding is not required. Only basic combat units are supported, 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



Player 0 units have a blue outline

Selected unit is highlighted in green

Barracks: train attack units

Heavy: high power but slow melee unit

Light: low power but fast melee unit. Heavy and Light units are very effective when micromanaged properly. Stay away from these units if you are not familiar with microRTS though.

Ranged: long range attack unit. Try to use this unit, as it is the most effective for human use.

Bases: accumulate resources and train workers

Workers: can harvest minerals and construct buildings

Minerals: harvest them with workers

Player 1 units have a red outline

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.