Heuristic Search Techniques for Real-Time Strategy Games

by

David Churchill

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) video games are known for being one of the most complex and strategic games for humans to play. With a unique combination of strategic thinking and dextrous mouse movements, RTS games make for a very intense and exciting game-play experience. In recent years the games AI research community has been increasingly drawn to the field of RTS AI research due to its challenging sub-problems and harsh real-time computing constraints. With the rise of e-Sports and professional human RTS gaming, the games industry has become very interested in AI techniques for helping design, balance, and test such complex games. In this thesis we will introduce and motivate the main topics of RTS AI research, and identify which areas need the most improvement. We then describe the RTS AI research we have conducted, which consists of five major contributions. First, our depth-first branch and bound build-order search algorithm, which is capable of producing professional human-quality build-orders in real-time, and was the first heuristic search algorithm to be used on-line in a STARCRAFT AI competition setting. Second, our RTS combat simulation system: SparCraft, which contains three new algorithms for unit micromanagement (Alpha-Beta Considering Durations (ABCD), UCT Considering Durations (UCT-CD) and Portfolio Greedy Search), each outperforming the previous state-of-the-art. Third, Hierarchical Portfolio Search for games with large search spaces, which was implemented as the AI system for the online strategy game Prismata by Lunarch Studios. Fourth, UAlbertaBot: our STARCRAFT AI bot which won the 2013 AIIDE STARCRAFT AI competition. And fifth: our tournament managing software which is currently used in all three major STARCRAFT AI competitions.

Acknowledgements

Will be added in camera-ready version.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Introduced in the early 1990s, real-time strategy (RTS) video games have recently become a popular test-bed for artificial intelligence research and application. Since Michael Buro's call-to-action paper [10] significant advancements have been made in RTS game AI with contributions from many fields within computer science and engineering. Motivation for RTS AI has also grown rapidly with the emergence of competitions such as the Open RTS (ORTS) AI competition, the Google AI Challenge (ANTS), and STARCRAFT AI Competitions (organized by AIIDE, CIG, and SSCAI).

1.1 Real-Time Strategy Games

Real-time strategy video games can be classified as strategic video games which simulate military warfare on various scales. Players assume the role of a military commander in charge of a group of forces which must build an economy (collect resources), construct a base (buildings and defenses), and establish a combat force (train units and research technologies) in order to defeat enemies by destroying their armies and bases. RTS games vary in size and complexity. However, they all share several traits which differ from traditional games:

Real-Time: RTS games are played in *real-time*, meaning that players can issue actions as fast as the game is executed (between 30 and 60 frames per second), and the game will progress normally even if no actions are given. This is unlike traditional games like CHESS or GO where

players may have several minutes to decide on an action, and the game cannot progress until a player has acted. For example, STARCRAFT runs at 24 frames per second, meaning that actions for each unit can be input once every 42ms.

Simultaneous Moves: In RTS games, more than one player can issue an action during the same time step. Additionally, these actions may be *durative*, i.e. requiring some time to complete.

Imperfect Information: Players in RTS games cannot see their opponent's units and actions unless they are actively scouting them. Typically, a map is initially covered by a *fog-of-war* which blocks vision of an area until it has been explored by the player.

Non-Determinism: Some RTS games have non-determinism in their actions. In STARCRAFT for example, units have a small chance to miss attacks if they are shooting from low ground to a target on high ground.

Multi-Unit Control: Most RTS games allow the user to control dozens of units at once, with each able to be given individual actions. This means that at any given state there may be an exponential number of possible actions with respect to the number of units a player controls.

Complexity: The complexity of RTS games is much higher than traditional games, in terms of state space size, the number of actions that can be performed at any time step, and the number of actions required to reach the end of a game. For example, the number of possible states in CHESS is approximately 10^{50}, GO has around 10^{170}, while STARCRAFT has shown to have at least 10^{1000} [59] as a very lenient lower bound.

1.2 Motivation

When STARCRAFT was released 1998, it captured the video game world in a way never seen before, with millions of players playing competitively over LAN and on Blizzard's battle.net servers. Not only did it sell millions of copies, but

it became so popular in South Korea that the Ministry of Culture, Sports and Tourism formed KeSPA, the Korean e-Sports association to manage and promote the professional play of STARCRAFT in the country. STARCRAFT has been played professionally in Korea and around the world ever since, with millions of dollars in prize money being awarded annually [35]. Top professional players have risen to celebrity status, with annual salaries topping several hundred thousand dollars paid by their teams and sponsors. Released in 2009, STARCRAFT II sold over one million copies on its first day, and has made competitive RTS games even more popular throughout the world. In 2013 there were more than a dozen RTS tournaments with prize pools over $50,000 [35]. With such an established industry and competitive RTS gaming scene we can motivate our research in artificial intelligence for real-time strategy games in several ways.

1.2.1 Creating Better AI Agents

Recent advances in traditional game AI have created computer programs (agents) which are capable of defeating the human world champions at several games such as Chess (Deep Blue), Checkers (Chinook), Othello (Logistello), and limit Texas Holdem poker (Cepheus). Competition, whether it is AI vs. humans, or AI vs. AI, has always been a motivating factor for research in the field of games AI, as it is in many fields. Advancing the state of the art in games AI also drives research which can be applied to other fields such as natural language processing and automated planning [10]. Powerful game AI programs have also had commercial success in the software industry as both entertainment and as training tools for players. For example, as Texas Holdem poker has become more popular in recent years, programs such as Poker Academy [6] employ AI techniques to help players train as well as keep track of individual statistics.

With video games, the application of strong AI has even more benefits than for traditional games. For both single-player and multiplayer video games, much of the gaming experience is often based on the interaction of the player character with computer controlled non-player characters and their environ-

ments. Long gone are the days of statically scripted side-scrollers where enemies appeared moving right-to-left in the same pattern every time the game was played, as gamers now demand more interaction and replayability from top titles. The reactions of players to the behaviour of these AI controlled characters is often the core of many video game reviews and critiques, with games such as FEAR and LEFT 4 DEAD having been praised for their advances in AI [60]. With the US video games industry doing over $65 billion in sales in 2011 and still growing, research into strong video game AI has much larger economic benefits than for traditional board games.

Most RTS games (including STARCRAFT) have been criticized specifically for their lack of challenging computer AI opponents. Due to the complexity of RTS AI, retail games have been restricted to implementing simply scripted behaviours which are easily exploited by human players. To compensate for a lack of skill, the programmers of these games often opt to give their AI unfair advantages such as complete map vision (WARCRAFT, STARCRAFT) or an economic advantage (STARCRAFT II insane AI difficulty mine minerals at a faster rate than humans). Even with these cheating tactics, humans still find ways to exploit their scripted behaviours, often able to beat up to 4 or 5 AI opponents at the same time. More advanced RTS AI would not only provide a better single player experience, but also provide good training partners for the ever growing field of eSports and professional gaming, which in itself is a multi million dollar industry.

1.2.2 RTS AI Competitions

RTS games have recently become popular within the AI research community due to their challenging properties. With the goal of eventually beating professional human players at popular RTS games like STARCRAFT, several RTS AI competitions have been created to foster competition and help improve the state-of-the-art. The first such competition took place in 2006 [11] with the development of the Open RTS (ORTS) [12] program at the University of Alberta. These competitions had several categories focusing on important sub-problems in RTS games such as resource gathering and small-scale combat.

With the release of the BroodWar Application Programming Interface (BWAPI) [39] in 2009, it became possible to control the popular retail game of STARCRAFT using C++ programs. In 2010 the first STARCRAFT AI Competition was organized by Ben Weber at the University of California, Santa Cruz as part of the AIIDE conference, and since 2011 it has been organized and run annually at the University of Alberta. Two other major STARCRAFT AI Competitions have arisen since then, namely the Computational Intelligence in Games (CIG) Competition, as well as the Student STARCRAFT AI Competition (SSCAI) [15]. These competitions have focused on playing the full game of STARCRAFT with no cheats or hacks allowed, bots must face the same harsh real-time conditions that human players face. These competitions have motivated many people to join the RTS AI community including both academics and hobbyists alike.

1.2.3 Game Design, Balance, and Testing

Whether it is a board game, video game, or sport, players will only be interested in playing it competitively if its rules are well balanced for all parties involved. RTS games typically give the player a choice of race (or faction) before starting a game, with each race offering different types of units, buildings, and play styles for the player to choose from. If one race offered significant advantage over another, there would be no incentive to choose any other race. Early RTS games avoided the need for balance by designing races symmetrically with the only major differences being in aesthetics. WARCRAFT II for example allowed players to play as Orcs or Humans, however all units (with the exception of one) had an identical counterpart within the other race, so players did not have a significant advantage by choosing either race.

When STARCRAFT was released, it featured 3 completely unique races: Terrans (human-like with a balance of unit types), Protoss (an advanced technological race with powerful expensive units), and Zerg (a bug-like race which focused on masses of inexpensive units). The complexity involved in balancing a game with three unique races was incredibly high, with new gameplay patches for STARCRAFT being released on a regular basis for almost 8 years.

By applying AI techniques to RTS games we can construct tools for automatic game balancing and play-testing, which can supplement human player feedback to more quickly find flaws in game design and help find better parameters for tuning game mechanics.

1.3 Thesis Outline

In chapter 2 we will decompose RTS AI into a number of sub-problems which have become their own areas of AI research in recent years. We will describe each sub-problem and how they related to each other, giving motivation and a brief literature survey of existing solutions for each topic. Chapters 3 to 6 will describe all research that has been performed for this thesis. Chapter 3 describes our Build-Order Search System (BOSS) for tackling the problem of build-order optimization in RTS games. BOSS is capable of finding build-orders in real time which are comparable to those of expert human players. Chapter 4 describes our research into RTS combat micromanagerment which produced three new algorithms: Alpha-Beta Considering Durations (ABCD), UCT Considering Durations (UCT-CD) and Portfolio Greedy Search, each of which outperformed the previous state-of-the-art in the field. We also introduce SparCraft: an open source combat project for simulating RTS game combat. Chapter 5 describes Hierarchical Portfolio Search (HPS): our algorithm for games with large state and action spaces which was implemented into the retail video game Prismata by Lunarch Studios. Chapter 6 will describe the open source software contributions that have been made as a result of our research, with the most notable being UAlbertaBot: our STARCRAFT AI competition entry. Finally, in chapter 7 we will conclude by summarizing the work presented in this thesis, and give directions for future research.

Chapter 2

RTS Sub-Problems, Background, and Literature Survey

Real-time strategy games are incredibly complex, even for professional human players. In order to manage this complexity, professional players have broken the game into several sub-problems which they can theorize about and practice individually. While these sub-problems are not truly independent strategically, it is a necessary abstraction in order to make the problem of playing such a complex game tractable for humans. Researchers have adapted this divide-and-conquer technique when approaching RTS AI, attempting to find solutions to various RTS decision sub-problems rather than tackle the game as a whole.

We can categorize types of RTS decision sub-problems based on both the time scale that the problem deals with, and the level of abstraction of the problem (Fig. 2.1). These categories **Strategy**, **Tactics**, and **Reactive Control**, are based on literature from both military command [90] and AI research [19]. Strategic problems involve the highest level decisions which determine your strategic stance and dictate orders at a global scale, while tactical problems typically involve smaller groups of units in an attempt to win battles or skirmishes in a more localized area. Andrew R Wilson, professor at the U.S. Naval War College says "Tactics and operations are about winning battles and campaigns. Strategy is about winning wars."[90] The third category, reactive control, involves computing concrete low-level unit actions.

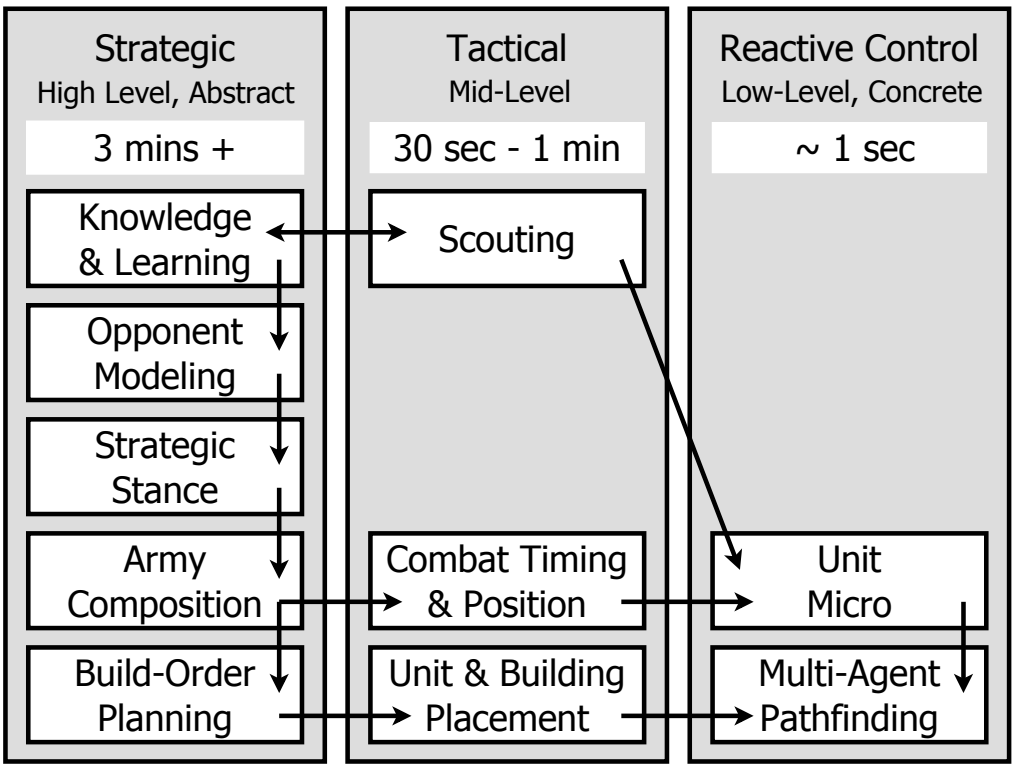


Figure 2.1: The main sub-problems in RTS AI research categorized by their approximate time scope and level of abstraction. Arrows indicate the direction that information flows hierarchically through the different sub-problems, similar to a military command structure.

These categories mimic a military command hierarchy, both in terms of chain of command as well as information processing. The higher level strategic commanders make broad global strategic decisions based on an abstract level of knowledge of troop movements and enemy capabilities. When a decision is made at the strategic level, an order is given to a tactical unit with only the information necessary to accomplish the tactical goal. These tactics are then carried out by individual troops employing their own form of reactive control at a low-level to accomplish individual tasks. Fig. 2.1 shows this RTS command hierarchy, as well as the flow of information within sub-problems.

2.1 Strategy

Strategy is the highest level of abstraction and corresponds to the most broad strategic decisions made by a player in an RTS game. Strategic decisions influence the game as a whole and rely on analyzing the long-term effects that actions can have later in a game. The current state of strategic decision making in both research and retail RTS AI rely heavily on hard-coded approaches with a large amount of expert knowledge.

Hard-coded (or scripted) approaches are by far the most common solution to strategic decision making, and most often take the form of finite state machines (FSM) to represent sets of strategic stances. These FSM systems abstract the game into high-level stances such as 'attacking', 'defending', or 'expanding' and then provide a set of hard-coded rules which trigger the system to enter a specific stance. For example: detecting that an enemy is in an aggressive stance may trigger the system to enter a defensive stance to stop that aggression. The stances often encode policies which cause lower-level tactical troop movements to be carried out. There are several benefits to these types of systems:

- They are easy to construct and intuitive to design
- They are well suited for incorporating expert domain knowledge
- They are computationally inexpensive

- They often produce results which are adequate for beginner-level AI for retail games.

The drawbacks to these systems, however, are:

- They cannot adapt to unforeseen situations that were not explicitly programmed, therefore many situations must be thought of by experts in order to produce a system which performs competitively
- They are not robust to game property changes and may require complete redesigns if the game changes significantly
- Their behaviours are often deterministic, repetitive, easily identifiable and exploitable by human players

While rule-based systems like FSMs and decision trees have had success in creating simple behaviours for modern retail video game AI, they still suffer from the drawbacks listed above and are not capable of producing expert human-level AI for more strategic video games like RTS. Because of this, more sophisticated and adaptable planning or learning techniques must be found. In the following sections we will introduce the sub-problems of RTS AI related to strategy and discuss existing solutions to these problems.

2.1.1 Knowledge and Learning

As with any game, all RTS players must have some knowledge of the game before playing. Examples of this type of knowledge can be game rules, unit properties, opening-book actions, and even some knowledge of their opponent. During play, players can gather and learn additional information about the game or about their opponents. This knowledge is used to guide all strategic aspects of play in an RTS game. Much of the related research to this topic is discussed in the following sub-sections.

2.1.2 Opponent Modeling and Prediction

In an RTS game, players typically start with no vision of their opponent's units or actions. With no knowledge of an opponent, the first few minutes

of an RTS game are a sort of rock-paper-scissors scenario in which players choose a strategic stance and implement it until they can observe the enemy through scouting. Players then attempt to learn a model of their opponent in order to predict their future actions and to choose their own actions to exploit perceived weaknesses.

Several directions have been chosen to attempt to model player and predict opponent strategies in RTS games. In [30], Dereszynski et al. used a Hidden Markov Model to learn build-order sequence probabilities of players in StarCraft, using this information to construct probabilistic player models. In [80] Synnaeve and Bessiere learned to predict opening game strategies from STARCRAFT replays used a semi-supervised Bayesian model. An important strategic element of RTS games is that of *tech*: the current level of research or technology prerequisites a player has met which dictates the type of units they are able to create. The graph of prerequisite tech in an RTS game is called a *tech tree*. In [81] they again used replay analysis coupled with use an unsupervised Bayesian learning model to predict which level of a tech tree an opponent was in. These prediction models were used in conjunction with another of their Bayesian modeling systems presented in [82] in BroodwarBotQ, their entry to the 2012 STARCRAFT AI Competition. They claimed that while their predictions were quite accurate, they placed a disappointing 4th in the competition due to their inability to adapt to the predictions effectively.

In [69], Schadd et al. applied Case-Based Reasoning (CBR) to a hierarchical structured model of an opponent in an attempt to classify an opponent as one of several predetermined labels (aggressive, defensive, tech, etc) in the SPRING RTS game (a clone of Total Annihilation). They concluded that they were able to accurately classify opponents using this method for these highly abstract labels. Kabanza et al. [45] analyzed the algorithmic challenges behind behaviour recognition in RTS games and proposed an architecture for helping to deal with several of the challenges, including encoding strategies as a hierarchical task network (HTN). Named the Hostile Intent Capability and Opportunity Recognizer (HICOR), their initial experiments showed promising results for strategy recognition in StarCraft. However, they assumed that their

agent had complete map knowledge, so further experiments with imperfect information need to be performed.

2.1.3 Strategic Stance

A player's strategic stance determines the *style* of play in an RTS game, which typically corresponds to a specific balance between combat aggression and economic expansion. One popular strategic stance is a *rush* in which combat units are constructed as fast as possible in order to throw the enemy off guard and win a quick victory. In contrast to this, *turtling* is a defensive stance which focuses on making static base defenses in order to hold off enemy forces while securing economic objectives. The choice of a particular strategic stance dictates the army composition a player wishes to obtain, and the time of the game when they wish to engage the enemy.

Several planning and learning based approaches have been developed to identify and choose these strategic stances. For example, in [19] Chung et al. use MCPlan - a Monte Carlo planning system for selecting abstract high level strategies in a simplified capture-the-flag (CTF) RTS game setting. By stochastically sampling the possible plans (explore, attack, move, etc) for a player, evaluating them, and then choosing the most statistically viable top-level plans, they were able to show promising results for this simplified CTF game. In [68], Sailer et al. perform strategy selection by approximating a Nash-equilibrium [55] over a set of high-level abstract strategies (attack, defend, move, etc) in a simplified RTS game. By comparing a Nash equilibrium player, a minimax player, and several single-scripted players they were able to conclude that the Nash and minimax players defeat the scripted players in this simple RTS game. However, no work was done to follow up for more complex RTS scenarios.

In [58], Ontañón et al. used real-time case-based planning (CBP) to learn plans from human demonstration in Wargus (a WarCraft 2 clone). Composed at run-time, these plans were then translated into overall strategies to play the entire game. Following up in [53], they improved on their work by incorporating situational assessment to improve the quality of the retrieved plans.

Aha et al. [1] also working in the Wargus domain used case-based reasoning (CBR) for dynamic plan retrieval in their Case-based Tactician (CaT) system which was successful in defeating scripted and evolved opponents.

Weber et al. [88, 86] used goal-driven autonomy and active behaviour trees to demonstrate a reactive planning framework for strategic and tactical goal selection in StarCraft. They demonstrated that the system performed in real-time within StarCraft, and that their EISBot improved in performance with the new system. However, they achieved only a 60% win rate against the built-in AI which is far below current competitive AI standards. Young and Hawes [92] used an evolutionary approach to prioritizing high level tasks which showed improvement over their statically scripted priorities for defeating the default in-game AI in StarCraft. Their system, however, only achieved a 68% win rate against the default AI, which should be close to 100% for any competitive STARCRAFT bot.

Miles [52] introduced IMTrees, in which each leaf node is an influence map and each intermediate node is an operation on those maps (addition, multiplication, etc). Using evolutionary algorithms to construct IMTrees, they were used for strategic decision making based on spatial reasoning on influence maps. Ontañón and Buro [57] propose an Adversarial Hierarchical Task Network (AHTN) which combines a minimax tree search with HTN planning with support for games with durative, simultaneous, and concurrent actions. Used for action selection in the simplified RTS game MICRORTS [56], AHTN outperforms Alpha Beta, UCT and Naive MCTS. With these promising initial results, they are working on extending it to more complex RTS games such as StarCraft. HTNs have also been applied to strategic decisions within simpler games such as first-person shooters [41].

In [5], Barriga et al. introduce Puppet Search: an adversarial search system which decides on which actions to take by searching over a set of tactical scripted behaviours. Given a set of these scripts, Puppet Search puts choice points within the scripts which then act as nodes in a search tree. Its search then produces a principal variation of scripts and choices which are to be executed by an agent for a given time period, which are more robust and

adaptable than the scripts themselves, being able to defeat a wider variety of opponents than any one of the scripts on its own.

2.1.4 Army Composition

Army composition is decided by strategic stance, with special consideration to the predicted opponent army composition. Each unit type in an RTS game has its own unique properties such as attack range/damage, armour type, hit points, speed, and whether or not it is a ground or flying unit. These complex interactions between unit types make army composition a difficult decision process in itself.

Certicky et al. [14] used case-based reasoning to select an army composition in StarCraft, based on their current observations of opponent units. While they said they had successful results they did not have any quantitative evaluation to support their claims, and concluded by saying that their method would have been more accurate if they had a better scouting system in their bot.

2.1.5 Build-Order Planning

Once an army composition has been chosen, the army units must be built by the player by first using worker units to gathering resources, then using these resources to construct additional buildings and infrastructure which can then produce the army units such as marines or tanks. The sequence of actions taken to arrive at a given set of goal units is called a build-order. Build-order planning can be described as a constraint resource allocation problem which features concurrent actions. The problem of build-order planning is mapped out by Kovarsky and Buro [48] which specify a simple build-order domain in PDDL, in an attempt to promote future research in the area. An evolutionary algorithm for finding STARCRAFT 2 build-orders was written by Brandy [8] and is able to find build-orders for a given goal set of units input by the user. However, due to the nature of evolutionary algorithms its running time is quite slow, and is unsuitable for constructing build-orders in real-time, though it could be used off-line to construct an *opening book* of build orders for use in an agent.

Previous solutions for build-order planning in RTS AI agents such as those present in the AIIDE STARCRAFT AI Competition have been a mix of hard-coded build-order sequences and priority based systems [59]. The most widely used system for planning build-orders for the AIIDE STARCRAFT AI Competition was the Broodwar Standard Add-on Library (BWSAL) for BWAPI, which provided a BuildOrderManager module. This module accepts prioritized build, upgrade, and research commands and then attempts to execute those commands such that higher priority items are executed first, with priorities being explicitly declared by the users. If the player does not currently have the prerequisites to build the highest priority items, BWSAL will determine the required actions and build them first with a higher priority. This system does not attempt to optimize the makespan of the resulting build-order or its resource cost, simply building prerequisites in order until the desired goal has been met, leaving a large burden on the player to decide which priorities will result in the lowest makespan. This system was widely used in STARCRAFT AI competitions from 2010 to 2012, but produced build-order plans of very low quality when compared to expert human players and most top AI agents replaced it with their own custom systems. Due to the lack of research in this area and the urgent need for a better real-time build-order planning system, my first research topic was to create a better build-order planning system, which resulted in the system represented in Chapter 3.

2.2 Tactics

Tactics are one step down from strategy, and are used as a means of securing strategic goals. Tactics are more spatially localized and generally focus on problems with a time scale of less than one minute. For human players, tactics often involve much more dexterity and the quick issuing of commands in game, as tactical problems typically deal with the movement of troops on the battlefield. In this section we will explain different tactical sub-problems and how they relate to the more abstract strategic problems.

2.2.1 Scouting

The act of information gathering and reconnaissance in RTS games is known as *scouting*. RTS maps are typically covered by an unobservable layer called a *fog-of-war* which disallows vision of areas of the map which aren't in the immediate vicinity of a player's units, adding imperfect information to the game. Each RTS unit has a unique vision radius, and enemy units can only be seen if they are within the vision radius of a friendly unit. This means that as in real combat, players must place their own units in harm's way in order to gain valuable information. In some RTS games, other methods of scouting can be used such as magical spells or technologies which temporarily reveal an area of the map, however these methods usually have an economic cost so their use is limited. Scouting is a key aspect of high-level play in order to continually provide information to the player to adjust strategic decisions.

In [87], Weber et al. use a particle filter model to estimate the position of enemy units in StarCraft. When an enemy unit entered the fog-of-war, they calculated its probable position based on its last known position and heading, updating their particle model in real-time. They claimed that their 2011 AIIDE STARCRAFT AI Competition entry EISBot improved its results against the in-game STARCRAFT AI as well as the 2010 AIIDE Competition bots by 10% when using the particle system over no enemy tracking system whatsoever. In [40] Hlady and Bulitko proposed a hidden semi-Markov models (HSMMs) as well as particle filters for unit tracking in first-person shooter games. They concluded that using HSMMs improved the accuracy of the created occupancy maps.

2.2.2 Combat Timing and Position

Once an army has been built, the player must decide where and when to attack an opponent. In RTS games, *attack timings* are vitally important to play due to the rock-paper-scissors nature of army compositions. For example, certain levels of technology must be obtained in an RTS game in order to train specific units or to research armor or weapon upgrades. Based on scouting

information, players may choose to attack an opponent at a time just before these critical technologies have been completed in order to gain a strategic advantage. Similarly, players may choose to attack an enemy while they are building an economic expansion, when they have a momentary lapse in defenses. Travel time must also be considered when deciding when to attack, as most competitive RTS maps require that player bases be separated by at least 20-30 seconds of travel time for an average unit. If a player's army has been scouted leaving to go on an attack, this time is vital for the defending player to prepare a defense.

Equally important to deciding when to attack is deciding where to attack. Abstract combat maneuvers such as flanking, grouping, and splitting are decided at the tactical level to match the type of attack being performed. For example, if an early-game attack occurs, the only means of attacking the enemy may be a single frontal assault on the enemy base, however a late game attack may involve waves of attacks on several areas of the map. Patrols of units may also be sent to explore the map in an attempt to catch enemy units which are away from their main base.

Most combat AI related literature (as in section 2.3.1) is focused on the lower-level problem of reactive control, rather than the more abstract task of planning when and where squads should attack in RTS games.

2.2.3 Building Placement

Building placement is a crucial part of RTS play, especially in the early game stages. Many different strategic and tactical decisions are made based on how buildings can be played on a given map. For example, production facilities can be placed near an enemy base if an early rush is to take place, or a defensive *wall-in* of your own base can be made with early structures if a defensive posture is to be taken. The tactical placement of structures can be extremely useful in stopping the advances of enemies in several ways, each based on the properties of the units available to the faction of the player. For example in Starcraft, the Terran race is able to lift its buildings off the ground once constructed, allowing a tight defensive wall to be formed at the entrance of a

base in order to halt the advance of enemy troops. This wall can then be lifted when the player needs to leave the base to attack, acting as a sort of drawbridge from a medieval castle. Players can also place structures to create a maze-like environment that the enemy must navigate before reaching the base, which when combined with static defenses create a deadly labyrinth for any incoming attacker in a similar style to the *tower-defense* style of video games, a tactic often deployed by the Zerg race in Starcraft. If an AI agent does not have the ability to place buildings in this manner, entire strategies which rely on them must be discarded, weakening the overall strength of the agent.

Most existing RTS AI agents employ a simple building placement strategy consisting of brute force searching a local area within their base until a legal building location is found. Certicky et al. [14] used Answer Set Programming (ASP) to attempt to solve the problem of walling off a base in order to better survive early enemy attacks. The paper, however, does not mention that there is a larger context to building placement other than simply creating a wall-type structure (such as preventing scouting, or optimizing economic layout) and concludes by saying other approaches are necessary to aid in overall building placement. Richoux et al [65] use a constraint optimization method which is able to determine whether or not a wall-in is possible for a given region in just a few milliseconds, showing promise for inclusion in future RTS AI agents. Barriga et al [4] use a genetic algorithm to optimize building locations given the result of combat simulations of waves of enemies attacking the base. Their results show that their method is able to greatly improve the defensive capabilities of an agent in a wide variety of environments, however it is not yet fast enough to be run in real-time.

2.3 Reactive Control

Reactive control problems involve carrying out specific actions on the unit level to accomplish tactical goals such as “Scout the enemy base” or “Defeat that squad of enemy units”.

2.3.1 Unit Micro

Micro is a term used in RTS games for the specific movements of units, usually in a combat-related context. Unit micro is incredibly important as it dictates at a low-level how units will move and attack to most efficiently achieve a goal. Professional STARCRAFT player Jaedong (who was known for being able to issue over 400+ actions per minute while playing) once said "That micro made me different from anyone else in Brood War, and I won a lot of games on that micro alone." RTS combat and micro problems are very difficult to solve due to the following properties:

- **Real-Time Control**: Players may issue commands to their units on every frame (42ms in STARCRAFT, 16ms in STARCRAFT II), so any delay in calculation can result in an advantage for your opponent.
- **Multiple Unit and Action Types**: RTS games may have dozens of different unit types, each having their own properties (such as hit points and attack strength) as well as unique abilities (such as attack or cast spells), making the rules for such a game very complex.
- **Simultaneous Moves**: Unlike traditional turn-based games, RTS video games allow both players to act at the same time.
- **Durative Actions**: Actions in RTS games have different durations. For example a Dragoon in STARCRAFT takes 28 frames to successfully attack a target whereas a Zergling takes only 6 frames. This results in one player possibly acting several times before another player gets another move.
- **Multi-Unit Control**: Unlike some games where a player moves one piece at a time, commands may be given to any number of player units that are currently able to act
- **High Branching Factor**: Due to large number of units in most RTS combat settings, and the fact that multiple units may be given actions at once, there can be an exponential number of actions possible at any

given state (e.g., all possible combinations of ways for your own units to target enemy units).

Balla and Fern [3] used the UCT [46] Monte Carlo tree search algorithm combined with game state abstractions to tackle the problem of tactical assault planning in Wargus. Their experiments were the simplest possible instance of Wargus combat involving only close range footman units with 4-directional movement. Their UCT algorithm did not run in real-time as they performed 5000 playouts in their evaluation, and in several of their experiments UCT was outperformed by the hard coded agents.

Kovarsky and Buro [47] introduced an algorithm called Randomized Alpha-Beta (RAB) to play a unit-targeting combat game with simultaneous moves and multi-unit control. They performed tests with two player agents: a Nash-equilibrium player which computes a depth-1 Nash-equilibrium strategy (depth limit due to computational limits), and the RAB algorithm. In order for alpha-beta to deal with simultaneous moves, each simultaneous move is approximated by a two-level sub-tree in which the first player to act is selected randomly at each move (hence random alpha-beta). They showed that the randomized alpha-beta player defeated a normal alpha-beta player, a Monte Carlo player, and the depth 1 Nash player. This work is the basis for my research in unit micromanagement which will be presented in Chapter 4.

Potential fields and influence maps have recently become popular for guiding unit movements in RTS games. Hagelbäck and Johansson [38] use potential fields for keeping units at a maximum firing range from their opponents in order to minimize incoming enemy fire. They again use potential fields [37] in the game TANKBATTLE as a means to deal with uncertainty and fog-of-war. Uriarte and Ontañón [84] use influence maps extensively to implement a "kiting" behaviour in their STARCRAFT bot Nova to avoid enemies from reaching their units. Avery et al. [2] use co-evolved influence maps enhanced with A* pathfinding to develop spatial tactics for a capture-the-flag scenario, finding that it was successful on generating CTF tactics for increasingly difficult maps. Smith et al. [73] then expand on this by testing to see if students could learn

spatial tactics more quickly by playing against scripted agents or those using a tactics based on co-evolved influence maps with spatial features.

Reinforcement Learning (RL) [79] has also been tried as a solution to small-scale RTS combat. Madeira et al. [50] suggest using expert domain knowledge to help RL methods learn faster in domains such as turn-based strategy games, while Jaidee and Muñoz-Avila [43] suggest to cut down the search space by learning one Q-function for each unit type, rather than each individual unit. Wender and Watson [89] implemented several different RL algorithms which performed small-scale STARCRAFT combat and found that of the algorithms tested, one-step Q-learning and Sarsa(λ) performed the best. Evolutionary algorithms have also been implemented for learning micro controllers. Ponsen and Spronck [63] used an evolutionary algorithm for combat in Wargus and found it was quickly able to develop good combat results in small scale combat. Othman et al. [61] show that evolutionary algorithms are quite good at optimizing parameter-driven combat micro systems, which is then used to implement a 2010 AIIDE micro competition bot.

2.3.2 Multi-Agent Pathfinding and Terrain Analysis

Pathfinding and terrain analysis are an important aspect of most video games, and is especially so in RTS games where a player may control dozens of units at once. RTS pathfinding typically consists of guiding multiple units on a 2-D map, with units having various properties such as size, speed, and acceleration. In most games, pathfinding focuses on shortest-path optimization, whereas RTS games may involve more complex optimizations involving unit damage, keeping units in formations, or avoiding enemy vision. In [33] Forbus et al. use geometric and pathfinding analysis to show the importance of having good spatial information in war-like game settings. The default pathfinding engine in the retail version of STARCRAFT uses the A* algorithm to construct a single path to a goal for all units in a unit group. The units in this group then jockey for position on this path, often getting stuck on each other resulting in units traveling in single file. Instead of arriving in force together, these single-file units can then be picked off one by one as they arrive at enemy territory.

Because of this, better pathfinding algorithms must be used to achieve strong tactical performance.

Due to the large computational needs of A*, it has been modified in many ways in order to produce pathfinding systems for real-time games. Hagelbäck [36] combined potential fields with A* pathfinding as a means of unit navigation in StarCraft, concluding that it was preferable to naive A* for STARCRAFT navigation. Demyen and Buro [29] developed an efficient triangulation-based pathfinding system which splits the game map into recursively more abstract triangulations, allowing for very fast pathfinding. This system combined with flocking behaviour [64, 83] is very similar to the triangulation-based pathfinding system in the retail version of STARCRAFT II. Sturtevant performed an extensive review of grid-based pathfinding methods in [78].

Written by Luke Perkins [62] the BroodWar Terrain Analyzer (BWTA) is a tool which is used by virtually every STARCRAFT AI Competition bot. BWTA provides terrain analysis functionality including region calculations based on merging Voronoi diagrams, and chokepoint detection. BWTA was released as an open source project and is included as part of the official BWAPI library.

Danielsiek et al. [28] used influence maps combined with a flocking behaviour to achieve intelligent movement of groups of units in Glest, an open source RTS game. They found that it achieved better results than either method individually and used it for unit pathfinding as well as for flanking enemy groups of units, concluding by saying that the method is highly dependent on parameter tuning for the individual game.

Chapter 3

Build-Order Optimization

The goal of any RTS game is to defeat the forces of your enemy, and in order to achieve that goal a player must first construct an army with which to fight. These armies must be built by the player by first using worker units to gathering resources, then using these resources to construct additional buildings and infrastructure, which can then produce an army of combat units such as soldiers or tanks. The sequence of actions taken to arrive at a given set of goal units is called a *build-order*. Professional human players learn or memorize several proven build-order sequences for the initial few minutes of a game, which they then later adapt on the fly based on information obtained about their opponent. There are two separate problems involved with arriving at a build-order: first, one must choose the goal - the set of units that are to form the army you wish to construct, and second, what economic actions to take in order to construct those units in the quickest or most resource efficient way possible. The content of this chapter is based on our publication [23] in which address the problem of finding time-optimal build-orders for a given set of goal units in a real-time setting. We do not consider the problem of how the goals are constructed. We evaluate our method by comparing the construction time (makespans) of these build-order sequences to those of professional STARCRAFT players. We call this system the Build-Order Search System (BOSS).

3.1 Background

The build-order optimization problem can be described as a constraint resource allocation problem with makespan minimization, which features concurrent actions. Because of their practical relevance, problems of this kind have been the subject of study for many years, predominantly in the area of operations research. [13] motivates research on build-order problems in the context of RTS games and proposes a way of modeling them in PDDL, the language used in the automated planning competitions. In [48] the issue of concurrent execution is studied in general and efficient action ordering mechanisms are described for the RTS game build-order domain. Existing techniques for build-order planning in the RTS game domain have focused mainly on the game WARGUS (an open source clone of WARCRAFT II), which is much simpler than STARCRAFT due to the limited number of possible actions and lower resource gathering complexity. Several of these techniques rely heavily on means-end analysis (MEA) scheduling. Given an initial state and a goal, MEA produces a satisficing plan which is minimal in the number of actions taken. MEA runs in linear time w.r.t. the number of actions in a plan, so it is quite fast, but the makespans it produces are often much longer than optimal.

Chan et al. [18] employ MEA to generate build-order plans, followed by a heuristic rescheduling phase which attempts to shorten the overall makespan. While they produce satisficing plans quite quickly, the plans are not optimal due to the complex nature of the rescheduling problem. In some cases they are able to beat makespans generated by human players, but do not mention the relative skill level of these players. This technique is extended in [17] by incorporating best-first search in an attempt to reduce makespans further by solving intermediate goals. They admit that their search algorithm is lacking many optimizations, and their results show that this is not only slower than their previous work but still cannot produce significantly better solutions. Branquinho and Lopes [9] extend further on these ideas by combining two new techniques called MeaPop (MEA with partial order planning) and Search and Learning A* (SLA*). These new results improve on the makespans generated

by MEA, but require much more time to compute, bringing it outside the range of real-time search. They claim to be investigating ways of improving the run-time of SLA*. These techniques however are only being applied to Wargus, with goals consisting of at most 5 types of resources. Interesting plans in STARCRAFT may involve multiple instances of up to 15 different units in a single goal and requiring far more workers, increasing complexity dramatically.

3.2 Build-Order Planning Model for Starcraft

Build-order planning in RTS games is concerned with finding a sequence of actions which satisfies a goal with the shortest makespan. It is our goal to use domain specific knowledge to limit both the branching factor as well as depth of search while maintaining optimality, resulting in a search algorithm which can run in real-time in a STARCRAFT playing agent. In STARCRAFT, a player is limited to a finite number of resources which they must both collect and produce throughout the game. All consumables (minerals, gas) as well as units (workers, fighters, buildings) are considered resources for the purpose of search. An action in our search is one which requires some type of resource, while producing another (combat actions are out of our scope). Resources which are used by actions can be of the forms Require, Borrow, Consume, and Produce [9]. Required resources, which are called prerequisites, are the ones which must be present at the time of issuing an action. A borrowed resource is one which is required, used for the duration of an action, and returned once the action is completed. A consumed resource is one which is required, and used up immediately upon issue. A produced resource is one which is created upon completion of the action.

Each action a has the form $a = (\delta, r, b, c, p)$, with duration δ (measured in game simulation frames), three sets of pre-conditions r (required), b (borrowed), c (consumed), and one set of produced items p. For example, in the STARCRAFT domain, the action $a =$ "Produce Protoss Dragoon" has $\delta = 600$, $r = \{\text{Cybernetics-Core}\}$, $b = \{\text{Gateway}\}$, $c = \{125 \text{ minerals}, 50 \text{ gas}, 2 \text{ supply}\}$, $p = \{1 \text{ Dragoon}\}$. States then take the form $S = (t, R, P, I)$, where t is the

current game time (measured in frames), vector R holds the state of each resource available (e.g. 2 barracks available, one currently borrowed until time X), vector P holds actions in progress but are not yet completed (ex: supply depot will finish at time X), and vector I holds worker income data (e.g. 8 gathering minerals, 3 gathering gas). Unlike some implementations such as [9], I is necessary due to abstractions made to facilitate search.

3.2.1 Abstractions

Without having access to the STARCRAFT game engine source code, it was necessary to write a simulator to compute state transitions. Several abstractions were made in order to greatly reduce the complexity of the simulation and the search space, while maintaining close to STARCRAFT-optimal results. Note that any future use of the term 'optimal' or 'optimality' refers to optimality within these abstractions:

We abstract mineral and gas resource gathering by real valued income rates of 0.045 minerals per worker per frame and 0.07 gas per worker per frame. These values have been determined empirically by analyzing professional games. In reality, resource gathering is a process in which workers spend a set amount of time gathering resources before returning them to a base. Although we fixed income rates in our experiments, they could be easily estimated during the game. This abstraction greatly increases the speed of state transition and resource look-ahead calculations. It also eliminates the need for "gather resource" type actions which typically dominate the complexity of build-order optimization. Due to this abstraction, we now consider minerals and gas to be a special type of resource, whose "income level" data is stored in state component I. Once a refinery location has been built, a set number of workers (3 in our experiments) will be sent to gather gas from it. This abstraction eliminates the need for worker re-assignment and greatly reduces search space, but in rare cases is not "truly" optimal for a given goal. Whenever a building is constructed, a constant of 4 seconds (96 simulation frames) is added to the game state's time component. This is to simulate the time required for a worker unit to move to a suitable building location within

Algorithm 1 Depth-First Branch & Bound

Require: goal G, state S, time limit t, bound b

```
1:  procedure DFBB(S)
2:      if TimeElapsed ≥ t then
3:          return
4:      if S satisfies G then
5:          b ← min(b, S_t)                          ▷ update bound
6:          bestSolution ← solutionPath(S)
7:      else
8:          while S has more children do
9:              S' ← S.nextChild
10:             S'.parent ← S
11:             h ← eval(S')                         ▷ heuristic evaluation
12:             if S'_t + h < b then
13:                 DFBB(S')
```

an arbitrary environment, since individual map data is not used in our search, but again could be estimated during the game.

3.2.2 Algorithm

We use a depth-first branch and bound algorithm to perform build-order search. The algorithm, which can be seen in Algorithm 1 takes a starting state S as input and performs a depth-first recursive search on the descendants of S in order to find a state which satisfies a given goal G. This algorithm has the advantage of using a linear amount of memory with respect to the maximum search depth. Since this is an any-time algorithm we can halt the search at any point and return the best solution so far, which is an important feature for real-time applications.

3.2.3 Action Legality

In order to generate the children of a state, we must determine which actions are legal in this state. Intuitively, an action is legal in state S if the simulation of the game starting in time will eventually produce all required resources without issuing any further actions. Given our abstractions, an action is therefore legal in state S if and only if the following conditions hold: 1) The prerequisites required or resources borrowed are either currently available, or being

created. Example: a Barracks is under construction, so fighter units will be trainable without any other actions being issued. 2) The consumed resources required by the action are either currently available or will be available at some point in the future without any other actions being taken. Example: we do not currently meet the amount of minerals required, however our workers will eventually gather the required amount (assuming there is a worker gathering minerals).

3.2.4 Fast Forwarding and State Transition

In general, RTS games allow the user to take no action at any given state, resulting in a new state which increases the internal game clock, possibly increasing resources and completing actions. This is problematic for efficient search algorithms since it means that all actions (including the null action) must be taken into consideration in each state of the game. This results in a search depth which is linear not in the number of actions taken, but in the makespan of our solution, which is often quite high. In order to solve this problem, we have implemented a *fast-forwarding* simulation technique which eliminates the need for null actions.

In STARCRAFT, the time-optimal build-order for any goal is one in which actions are executed as soon as they are legal, since hoarding resources cannot reduce the total makespan. Although resource hoarding can be a vital strategy in late-game combat, it is outside the scope of our planner. Let us define the following functions:

- $S' \leftarrow \text{Sim}(S, \delta)$: Simulate the natural progression of a STARCRAFT game from a state S through δ time steps given that no other actions are issued, resulting in a new state S'. This simulation includes the gathering of resources (given our economic abstraction) and the completion of durative actions which have already been issued.

- $\delta \leftarrow \text{When}(S, R)$: Takes a state S and a set of resource requirements R and returns the earliest time δ for which $\text{Sim}(S, \delta)$ will contain R. This

function is typically called with action prerequisites to determine when the required resources for an action a will be ready.

- $S' \leftarrow \text{Do}(S, a)$: Issue action a in state S assuming all required resources are available. The issuing of the action involves subtracting the consume resources, updating actions in progress and flagging borrowed resources in use. The resulting state S' is the state for which action a has just been issued and has its full duration remaining.

$$S' = \text{Do}(\text{Sim}(S, \text{When}(S, a)), a)$$

now defines our state transition function which returns the state S' for which action a has been issued.

3.2.5 Concurrent Actions and Action Subset Selection

A defining feature of RTS games is the ability to perform concurrent actions. For example, if a player has a sufficient amount of resources they may begin the concurrent construction of several buildings as well as the training of several units. In a general setting, this may cause an action-space explosion because a super-exponential number of possible actions sequences has to be considered. Even in the common video game setting in which a game server sequentializes incoming concurrent player actions, it can be co-NP hard to decide whether these actions when sequentialized in arbitrary order result in the same state [13]. Fortunately, many RTS games, including STARCRAFT, have the property that simultaneously executable actions are independent of each other, i.e. action effects don't invalidate prerequisites of other actions: For any two actions a, b to be executable concurrently in state S we must have $\delta = \text{When}(S, \text{prerequisites of } a \text{ and } b) = 0$, which means $\text{Sim}(S, \delta) = S$. Because function $\text{Do}(S, x)$ returns a state in which pre-condition resources are decreased and post-condition resources are increased, we have

$$\begin{aligned} \text{Do}(\text{Do}(S, a), b) &= \text{Do}(S, a + b) \\ &= \text{Do}(\text{Do}(S, b), a), \end{aligned}$$

where '+' indicates the concurrent issuing of two actions, proving that the ordering of concurrent actions has no effect on the resulting state. We can also apply this argument iteratively for subsets larger than two actions. Based on this insight and the "earliest execution" property of optimal action sequences we discussed in the previous subsection, we can therefore impose a single ordering on simultaneous actions to eliminate the need for iterating over all possible sequences of concurrent actions from a given state.

3.2.6 Heuristics and Macro Actions

Our depth-first branch and bound algorithm allows us to prune nodes based on heuristic evaluations of the path length left to our goal. Line 12 of Algorithm 1 shows that we can prune a child node if its length so far plus its heuristic evaluation is less than the upper bound. If our heuristic is admissible, this guarantees that our computed solution will be optimal. We use the following admissible lower-bound heuristics to prune our search:

- LandmarkLowerBound(S,G): STARCRAFT's tech tree imposes many prerequisites on actions. These actions are known in the search literature as landmarks. Given this sequence of non-concurrent landmark actions, we sum the individual durations of actions not yet created to form an admissible lower bound for our search.

- ResourceGoalBound(S,G): Summing the total consumed resource cost of units in a goal gives us a lower bound on the resources required to construct the goal optimally. Performing a quick search to determine the makespan of producing only these resources is an admissible heuristic.

We can then take the maximum of these three heuristics as our heuristic value h. The heuristic used as an upper bound for our search is TrivialPlan(S,G)

— Given a state and a goal, we simply take a random legal action from the goal and issue it when it is possible. This guarantees that our goal is met, but does not optimize for time. The length of this plan is then used as an upper bound in our search.

To limit the branching factor of our search, we impose upper bounds on certain actions. E.g. if our goal contains two fighter units which are trained at a barracks, we know that we need to produce at most two barracks. Since it is difficult to pre-compute the optimal number of worker and supply units for a given goal in this fashion, higher bounds are placed on them to give a higher chance that optimal numbers can be produced.

Macro actions (also called *options* in reinforcement learning) have proven useful in speeding up search and planning through incorporating domain specific knowledge [42]. While these actions can be learned [77], we have simply hand-created several macro actions by inspecting build-orders used by professional players. Our macros all take the form of *doubling* existing actions which are commonly executed in sequence. For example: professional players often build worker or fighter units in bunches, rather than one at a time. By creating macro actions such as these we cut the depth of search dramatically while maintaining close to time-optimal makespans. To implement this, for each action we associate a repetition value K so that only K actions in a row of this type are allowed. The effects of macro actions can be seen in Fig. 3.1.

3.3 Experiments

Experiments were conducted to compare build-orders used by professional STARCRAFT players to those produced by our planner. Although our planner is capable of planning for each race, we limited our tests to Protoss players in order to avoid any discrepancies caused by using build-orders of different races. 100 replays were chosen from various repositories online, 35 of which feature professional players Bisu, Stork, Kal, and White-Ra. The remaining replays were taken from high level tournaments such as World Cyber Games.

The BWAPI STARCRAFT programming interface was used to analyze and

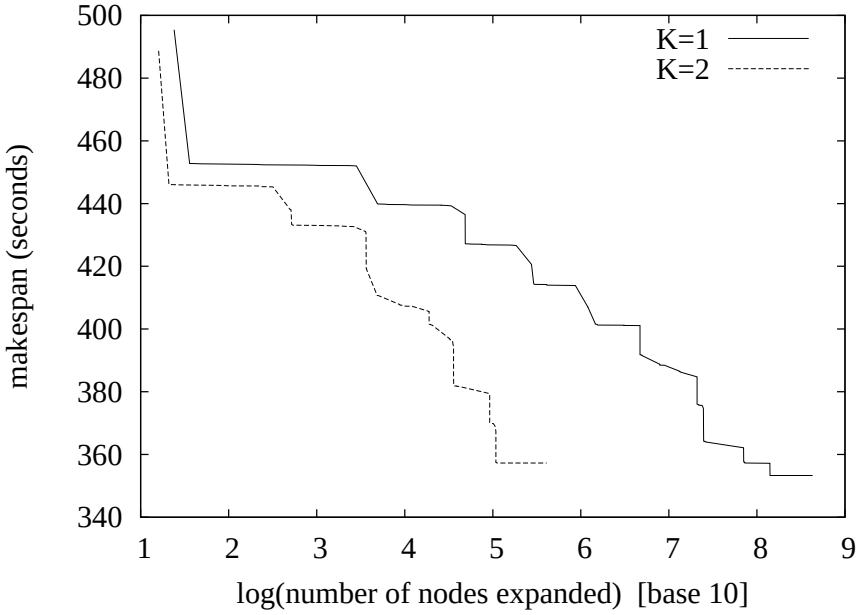


Figure 3.1: Makespan vs. nodes searched for late-game goal of two carriers, comparing optimal search ($K = 1$) and approximate search with macro actions ($K = 2$). Macro actions make complex searches tractable while maintaining close to optimal makespans.

extract the actions performed by the professional players. Every 500 frames (21s) the build-order implemented by the player (from the start of the game) was extracted and written to a file. Build-orders were continually extracted until either 10000 frames (7m) had passed, or until one of the player's units had died. A total of 520 unique build-orders were extract this way. We would like to have used more data for further confidence, however the process of finding quality replays and manually extracting the data was quite time consuming. Though our planner is capable of planning from any state of the game, the beginning stages were chosen as it was too difficult to extract meaningful build-orders from later points in the game due to the on-going combat. To extract goals from professional build-orders, we construct a function GetGoal(B,t_s,t_e) which given a professional build-order sequence B, a start time t_s and an end time t_e computes a goal which contains all resources produced by actions issued in B between t_s and t_e.

Tests were performed on each build-order with the method described in Algorithm 2 with both optimal (opt) and macro action (app) search. First

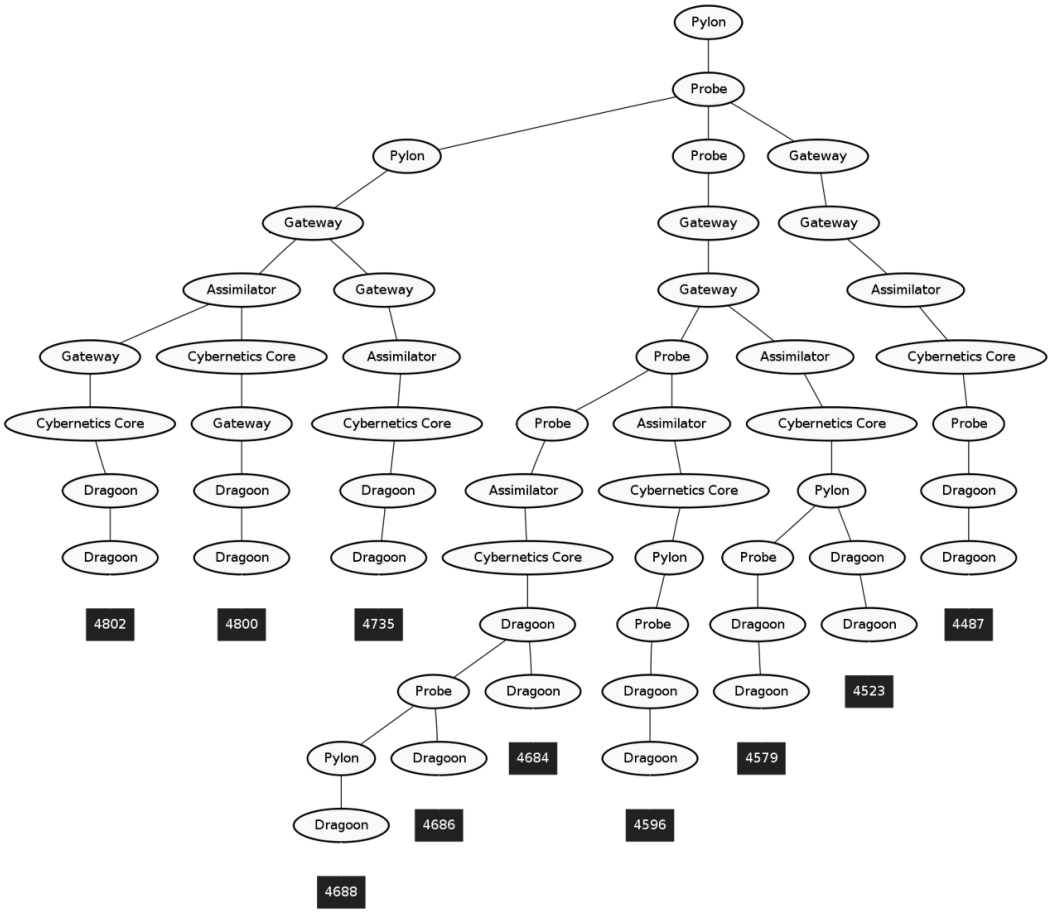


Figure 3.2: A sample search episode of BOSS applied to STARCRAFT using the Protoss race, starting with 8 Probes and 1 Nexus, with the goal of building two Dragoon units in the quickest way possible. The left-most path is the first build-order found by algorithm 1 which satisfies the goal (makespan listed below in STARCRAFT game frames). Each other leaf from left to right represents the final node of a build-order which has a new shortest makespan, with the shortest build-order being the right-most path. This figure clearly demonstrates the any-time nature of the algorithm, as it can stop at any point and and return the best solution found so far.

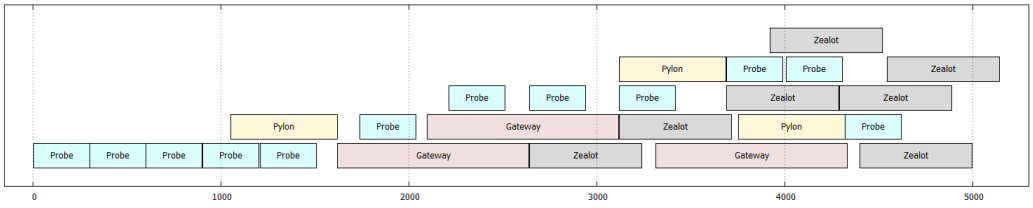


Figure 3.3: Concurrency chart for a build-order produced by BOSS with a goal of 7 Protoss Zealot units. X-axis measured in STARCRAFT game frames.

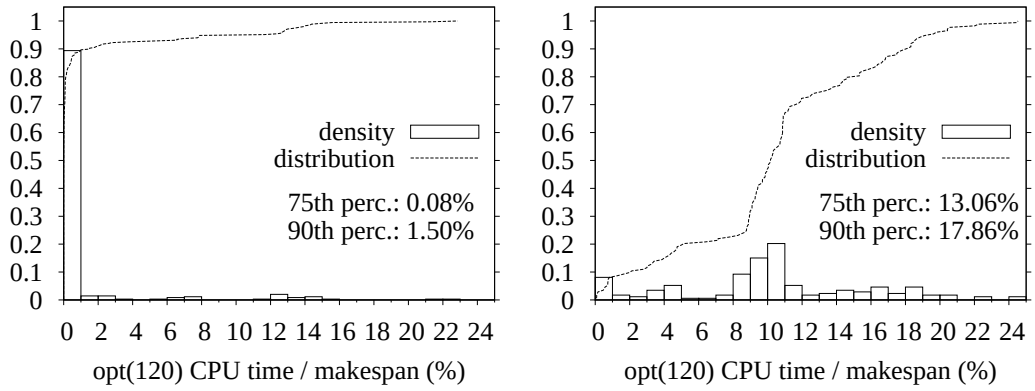
Algorithm 2 Compare Build-Order

Require: BuildOrder B, TimeLimit t, Increment Time i

```
1: procedure CompareBuildOrder(B,t,i)
2:     S ← Initial STARCRAFT State
3:     SearchPlan ← DFBB(S,GetGoal(B, 0, ∞),t)
4:     if SearchPlan.timeElapsed ≤ t then
5:         return MakeSpan(SearchPlan) / MakeSpan(B)
6:     else
7:         inc ← i
8:         SearchPlan ← ∅
9:         while inc ≤ MakeSpan(B) do
10:            IncPlan ← DFBB(S,GetGoal(B,inc-i,inc),t)
11:            if IncPlan.timeElapsed ≥ t then
12:                return failure
13:            else
14:                SearchPlan.append(IncPlan)
15:                S ← S.execute(IncPlan)
16:                inc ← inc + i
17:        return MakeSpan(SearchPlan) / MakeSpan(B)
```

with $t = 60s$ and $i = 15s$, second with $t = 120s$ and $i = 30s$. This incremental tactic is believed to be similar in nature to how professionals re-plan at various stages of play, however it is impossible be certain without access to professionally labeled data sets (for which none exist). We claim that build-orders produced by this system are "real-time" or "online" since they consume far less CPU time than the durations of the makespans they produce. Agents can implement the current increment while it plans the next. It should be noted that this experiment is indeed biased against the professional player, since they may have changed their mind or re-planned at various stages of their build-order. It is however the best possible comparison without having access to a professional player to implement build-orders during the experiment. Figs. 3.4 (time statistics), 3.5 and 3.6 (makespan statistics) display the results of these experiments, from which we can conclude our planner produces build-orders with comparable makespans while consuming few CPU resources. Results for 60s incremental search were similar to 120s (with less CPU usage).

A) CPU time statistics for search without macro actions:



B) CPU time statistics for search with macro actions:

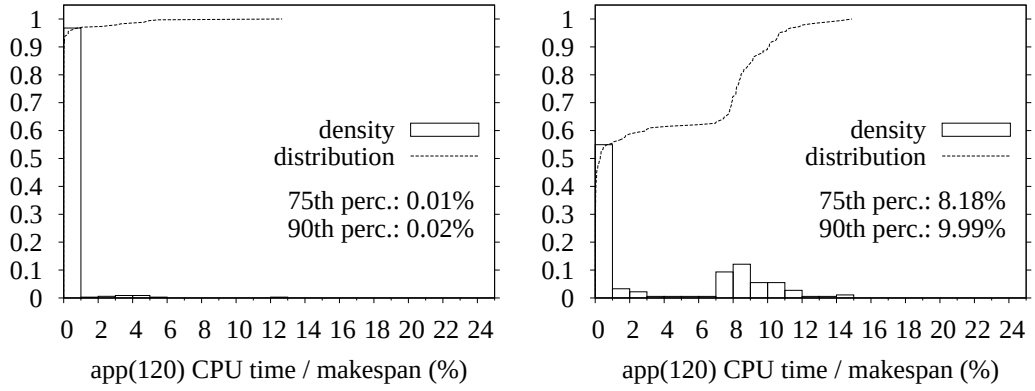


Figure 3.4: CPU time statistics for search without (A), and with (B) macro actions at 120s increments. Shown are densities and cumulative distributions of CPU time/makespan ratios in % and percentiles for professional game data points with player makespans 0..249s (left) and 250..500s (right). E.g. the top-left graph indicates that 90% of the time, the runtime is only 1.5% of the makespan, i.e. 98.5% of the CPU time in the early game can be used for other tasks. We can see that macro actions significantly reduce CPU time usage for build-orders with longer makespans.

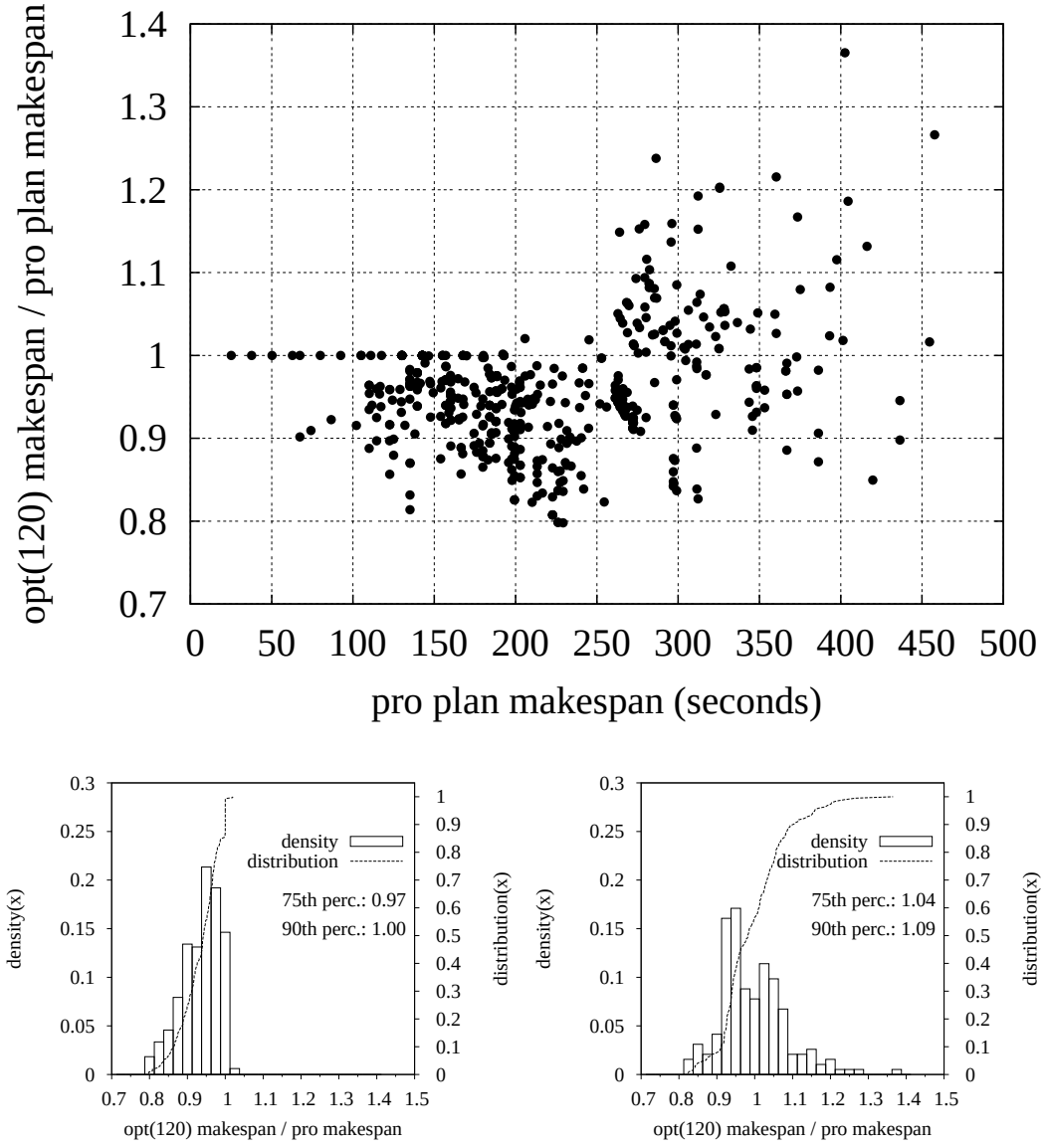


Figure 3.5: Makespan statistics for search **without** macro actions. Goals extracted by looking ahead 120s relative to professional player plan makespans. Shown are scatter plots of the makespan ratios (top), ratio densities, cumulative distributions, and percentiles for early game scenarios (pro makespan 0..249s, bottom left) and early-mid game scenarios (250..500s, bottom right). E.g. the top-middle graph indicates that 90% of the time, our planner produces makespans that match those of professionals

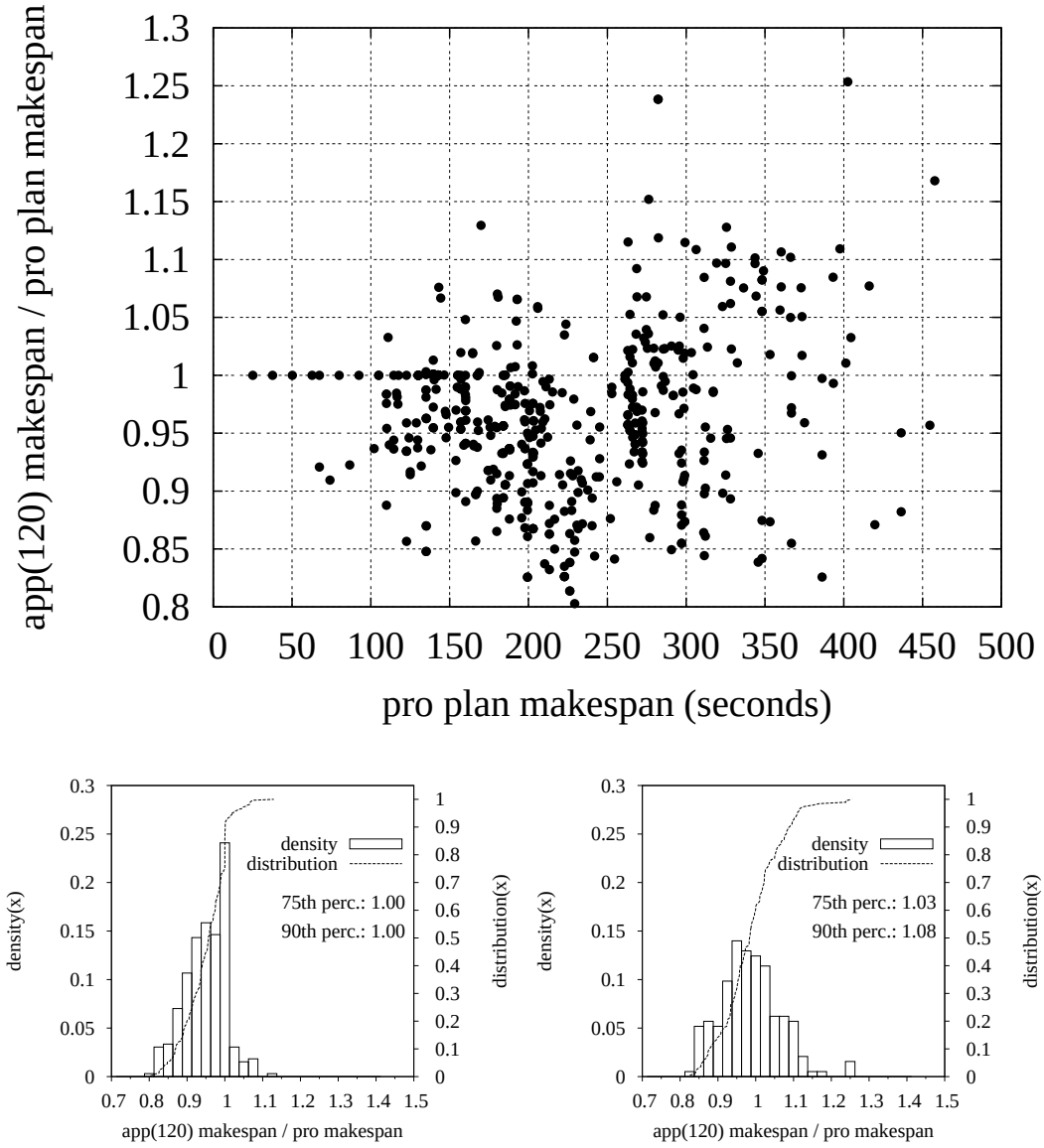


Figure 3.6: Makespan statistics for search **with** macro actions. Shown are scatter plots of the makespan ratios (top), ratio densities, cumulative distributions, and percentiles for early game scenarios (pro makespan 0..249s, bottom left) and early-mid game scenarios (250..500s, bottom right). We can see that macro actions slightly increase makespans for short build-orders, while slightly reducing makespans for longer build-orders.

3.4 Summary

In this chapter we have presented our Build-Order Search System (BOSS), a collection of heuristics and abstractions that reduce the search effort for solving build-order problems in STARCRAFT significantly while producing near optimal plans in real-time. We have shown macro actions, breadth limiting techniques, income abstractions, and multiple lower bound heuristics which reduce search spaces exponentially. A fast forwarding approach was introduced which replaced the null action, cut down on simulation time, and eliminated the need to solve the subset action selection problem. We have shown that with all of these techniques, our planner is capable of producing plans in real-time which are comparable to professional STARCRAFT players, many of which have played the game for more than 10 years. BOSS has been released as an open source software project [20], as well as being incorporated into UAlbertaBot, our STARCRAFT AI agent which won the 2013 AIIDE STARCRAFT AI Competition and is described in detail in section 6.1.

Chapter 4

RTS Combat Micromanagement

Unit micromanagement in RTS games (often called Micro) describes the problem of issuing commands to units while in combat in order to most effectively fight a group of enemy units, and is an incredibly complex and important part of RTS gameplay (see sub-section 2.3.1). This chapter will be comprised of several subsections summarizing the results of two of our related publications on RTS combat: In [26] we introduced Alpha-Beta Considering Durations (ABCD), a modification of the traditional Alpha-Beta algorithm for use in games with simultaneous and durative actions. In [24] we introduced two new algorithms: UCT Considering Durations (UCT-CD), a modification of the UCT Monte Carlo Tree Search algorithm for games with simultaneous and durative actions, as well as a new hill-climbing algorithm called Portfolio Greedy Search. This publication won the Best Student paper award at the 2013 conference on Computation Intelligence in Games (CIG) and was selected as an invited talk for the AI Summit of Game Developer's Conference (GDC) 2014. We will begin the chapter by discussing our SparCraft system for simulating STARCRAFT combat which facilitates research in this area.

4.1 Modeling RTS Combat: SparCraft

In order to perform search for combat scenarios in STARCRAFT, we must construct a system which allows us to efficiently simulate the game itself. The BWAPI programming interface [39] allows for interaction with the STARCRAFT game engine, but unfortunately, it can only run the engine at 32 times normal speed and does not allow us to create and manipulate local state instances efficiently. As one search may simulate millions of moves, with each move having a duration of at least one simulation frame, it remains for us to construct an abstract model of STARCRAFT combat which is able to efficiently implement moves in a way that does not rely on simulating each in-game frame. In the this section, we will discuss the SparCraft system we have constructed which allows for an abstract simulation of STARCRAFT Combat.

SparCraft was designed to be easily integrated into BWAPI based STARCRAFT AI bots. It includes:

- A STARCRAFT combat simulation system that uses BWAPI to access all game data such as unit and weapon properties
- An OpenGL tool for visualizing simulated combat scenarios
- A modular system for easily implementing custom combat AI behaviour
- Several state of the art combat AI algorithms including Alpha-Beta, UCT, and PortfolioGreedySearch (which will be discussed in the following sections)

In SparCraft, units can be given attack, move, and wait commands. All unit properties such as hit points, cool-down period, speed, size, armour, and weapon types are modeled exactly from STARCRAFT with the exception of acceleration, with all units having constant speed while moving. All upgrades and research are modeled. However, spell casters and units that contain other units (reavers, carriers, bunkers, transports) are not yet implemented. SparCraft does not yet implement unit collisions (to increase simulation speed) or fog of war by design in order to trade some simulation accurary for speed.

The combat model of SparCraft is comprised of three main data components and two main logic functions:

State $s = \langle \mathrm{t}, U_1, U_2 \rangle$

- Current game time t
- Sets of units U_i under control of player i

Unit $u = \langle p, \mathrm{hp}, \mathrm{t}_a, \mathrm{t}_m, \mathrm{type} \rangle$

- Position $p = \langle x, y \rangle$ in $\mathbb{R}^2$
- Current hit points hp
- Time step when unit can next attack t_a, or move t_m
- STARCRAFT unit type, defining all static unit properties such as damage, maximum hp, armor, speed, etc

Move $m = \langle a_1, \ldots, a_k \rangle$, a set of unit actions $a_i = \langle u, \mathrm{type}, \mathrm{target}, t \rangle$, with

- Unit u to perform this action
- The type of action to be performed: *Attack* unit target, *Move* u to position target, or *Wait* until time t

Player function p $[m = p(s, U)]$

- Input state s and units U under player's control
- Performs Move decision logic
- Returns move m generated by p

Game function g $[r = g(s, p_1, p_2)]$

- Initial state s and players p_1, p_2
- Performs game simulation logic
- Returns game result r (win, lose or draw)

Given a state s containing unit u, we generate legal unit actions as follows: if $u.\mathrm{t}_a \leq s.\mathrm{t}$ then u may *attack* any target in its range, if $u.\mathrm{t}_m \leq s.\mathrm{t}$ then u may *move* in any legal direction, if $u.\mathrm{t}_m \leq s.\mathrm{t} < u.\mathrm{t}_a$ then u may *wait* until $u.\mathrm{t}_a$. If both $u.\mathrm{t}_a$ and $u.\mathrm{t}_m$ are $> s.\mathrm{t}$ then a unit is said to have no legal actions. A legal player move is then a set of all combinations of one legal unit action from each unit a player controls.

Unlike strict alternating move games like CHESS, our model's moves have durations based on individual unit properties. We define the player to move next as the one which controls the unit with the minimum time for which it can attack or move. This means that at any given state a move may be able to be performed by either player, both players, or no player at all. Based on this model we can implement a fast-forwarding approach in which game frames between actions are skipped, avoiding unnecessary computations. Using this implementation SparCraft can simulate several million unit actions per second, allowing for the algorithms described in the following sections to be performed in real-time. For full documentation of the SparCraft package please see the SparCraft Google code wiki in [21]. In the following sections we will discuss the algorithm research which was done using SparCraft as a simulation engine.

4.2 Solution Concepts for Combat Games

The combat model defined in section 4.1 can naturally be complemented with a termination criterion and utility functions for the players in terminal positions. A position is called *terminal* if all the units of a player have reached 0 hp, or if a certain time limit (measured in game frames, or unit actions) has been exceeded. Combining the combat model with the termination criterion and utility functions defines a class of games we call *combat games*. In what follows we will assume that combat games are zero-sum games, i.e., utilities for both players add up to a constant across all terminal states. A single step simultaneous move game (such as ROCK, PAPER, SCISSORS) with action sets A_1 and A_2 can be classified as a *matrix game*. Each entry in the matrix A_{rc} is a *payoff* corresponding to player one choosing the action in row r of the ma-

trix and player two choosing the action in column c. Two player simultaneous move games with more than one step are often called *stacked matrix games*, as at every state there is an action combination which either leads to a terminal state, or to a subgame which is also a stacked matrix game. The properties of combat games together with simultaneous moves and fully observable state variables places combat games in this class of stacked matrix games. Such games can — in principle — be solved by backward induction starting with terminal states via Nash equilibrium computations for instance by solving linear programs [67]. However Furtak and Buro [34] showed that deciding which player survives even in combat games *without* movement is PSPACE-hard in general. This means that no known polytime algorithms exist for optimally playing combat games, and that in practice we have to resort to approximations. There are various ways to approximate optimal play in combat games. In the following sub-sections we will discuss a few of them.

4.2.1 Scripted Behaviours

The simplest approach, and the one most commonly used in video game AI systems, is to define static behaviors via AI scripts. Their main advantage is computation speed, but scripts often lack foresight, which makes them vulnerable against search-based methods. Scripted solutions are often used by retail video games and by bots in the STARCRAFT AI competitions [59], with behaviours similar to those implemented by humans in competitive games. We have implemented the following scripted behaviours as part of SparCraft:

- **Random**: Picks legal moves with uniform probability.
- **Attack-Closest**: Units will attack the closest opponent unit within its weapon range if it can currently fire. Otherwise, if it is within range of an enemy but is reloading, it will wait in-place until it has reloaded. If it is not in range of any enemy, it will move toward the closest enemy a fixed distance.
- **Attack-Weakest**: Similar to Attack-Closest, except units attack an opponent unit with the lowest hp within range when able.

- **Kiting**: Similar to Attack-Closest, except it will move a fixed distance away from the closest enemy when it is unable to fire.

- **Attack-Value**: Similar to Attack-Closest, except units attack an enemy unit u with the highest $\text{dpf}(u)/\text{hp}(u)$ value within range when able. This choice leads to optimal play in 1 vs. n scenarios [34].

- **NOK-AV**: (No-OverKill-Attack-Value) strategy is similar to Attack-Value, except units will not attack an enemy unit which has been assigned lethal damage this round already. It will instead choose the next priority target, or wait if one does not exist.

- **Kite-AV**: Similar to Kiting, except it will choose an attack target similar to Attack-Value.

4.2.2 Game Theoretic Approximations

As previously mentioned, combat games fall into the class of two-player zero-sum simultaneous move games. If we concentrate on the battle and define a zero-sum utility function, we can leverage many results from game theory. In this setting, the concepts of optimal play and game values are well defined, and the Nash equilibrium value $\text{Nash}(G)$ of a game G (in view of the maximizing player MAX) can be determined by using backward induction. However, as discussed earlier, this process can be very slow. Kovarsky and Buro [47] describe how games with simultaneous moves can be sequentialized to make them amenable to fast Alpha-Beta tree search, trading optimality for speed.

The idea is to replace simultaneous move states by two-level subtrees in which players move in turn, maximizing respectively minimizing their utilities. The value of the sequentialized games might be different from $\text{Nash}(G)$ and it depends on the order we choose for the players in each state with simultaneous moves: If MAX chooses his move first in each such state, the value of the resulting game we call the *pure maxmin value* and denote it by $\text{mini}(G)$. Conversely, if MAX gets to choose after MIN, we call the game's value the *pure minmax value* (denoted $\text{maxi}(G)$). An elementary game theory result is

that pure minmax and maxmin values are bounds for the true game value.

Proposition 1 *For stacked matrix games G, we have* $\text{mini}(G) \leq \text{Nash}(G) \leq \text{maxi}(G)$*, and the inequalities are strict iff the game does not admit optimal pure strategies.*

It is possible that there is no optimal pure strategy in a game with simultaneous moves, as ROCK-PAPER-SCISSORS proves. Less intuitively so, the need for randomized strategies also arises in combat games, even in cases with 2 vs. 2 immobile units ([34]). To mitigate the potential unfairness caused by the Minmax and Maxmin game transformations, Kovarsky and Buro [47] propose the Random-Alpha-Beta (RAB) algorithm. RAB is a Monte Carlo algorithm that repeatedly performs Alpha-Beta searches in transformed games where the player-to-move order is randomized in interior simultaneous move nodes. Once time runs out, the move with the highest total score at the root is chosen.

In [47], Kovarsky and Buro show that RAB can outperform Alpha-Beta search on the Maxmin-transformed tree, using iterative deepening and a simple heuristic evaluation function. In our experiments, we will test the stripped down RAB version we call *RAB'*, which only runs Alpha-Beta once. Another approach of mitigating unfairness is to alternate the player-to-move order in simultaneous move nodes on the way down the tree. We call this tree transformation *Alt*. Because RAB' and the Alt transformation just change the player-to-move order, the following result on the value of the best RAB move ($\text{rab}(G)$) and Alt move ($\text{alter}(G)$) are easy to prove by induction on the tree height:

Proposition 2 *For stacked matrix game G, we have*

$$\text{mini}(G) \leq \text{rab}(G), \text{alter}(G) \leq \text{maxi}(G)$$

The proposed approximation methods are much faster than solving games by backward induction. However, the computed moves may be inferior. This method of using search to generate move sequences has considerable advantages over scripted behavior, as anybody who tried to write a good rule-based CHESS program can attest:

- Search naturally adapts to the current situation. By looking ahead it will often find winning variations, where scripted solutions fail due to the enormous decision complexity. For example, consider detecting mate-in-3 situations statically, i.e. without enumerating move sequences.

- Creating search-based AI systems usually requires less expert knowledge and can therefore be implemented faster. Testament to this insight is Monte Carlo tree search, a recently developed sample based search technique that revolutionized computer Go [27].

4.3 Fast Search Methods for Combat Games

In the previous section we discussed multiple game transformations that would allow us to find solutions by using backward induction on stacked matrix games. However, when playing RTS games the real-time constraints are harsh. Often, decisions must be made during a single simulation frame, which can be 50 ms or shorter. Therefore, computing optimal moves is impossible for all but the smallest settings and we need to settle for approximate solutions: we trade optimality for speed and hope that the algorithms we propose defeat the state of the art AI systems for combat games. The common approach is to declare nodes to be leaf nodes once a certain depth limit is reached. In leaf nodes MAX's utility is then estimated by calling an *evaluation function*, and this value is propagated up the tree like true terminal node utilities. In the following subsections we will first adapt the Alpha-Beta search algorithm to combat games by handling durative moves explicitly and then present a series of previously known and new evaluation functions.

4.3.1 Simultaneous Move Sequentialization

Consider Fig. 4.1 which displays a typical path in the sequentialized game tree. Because of the weapon cooldown and the space granularity, battle games exhibit numerous *durative moves*. Indeed, there are many time steps where the only move for a player is just pass, since all the units are currently unable

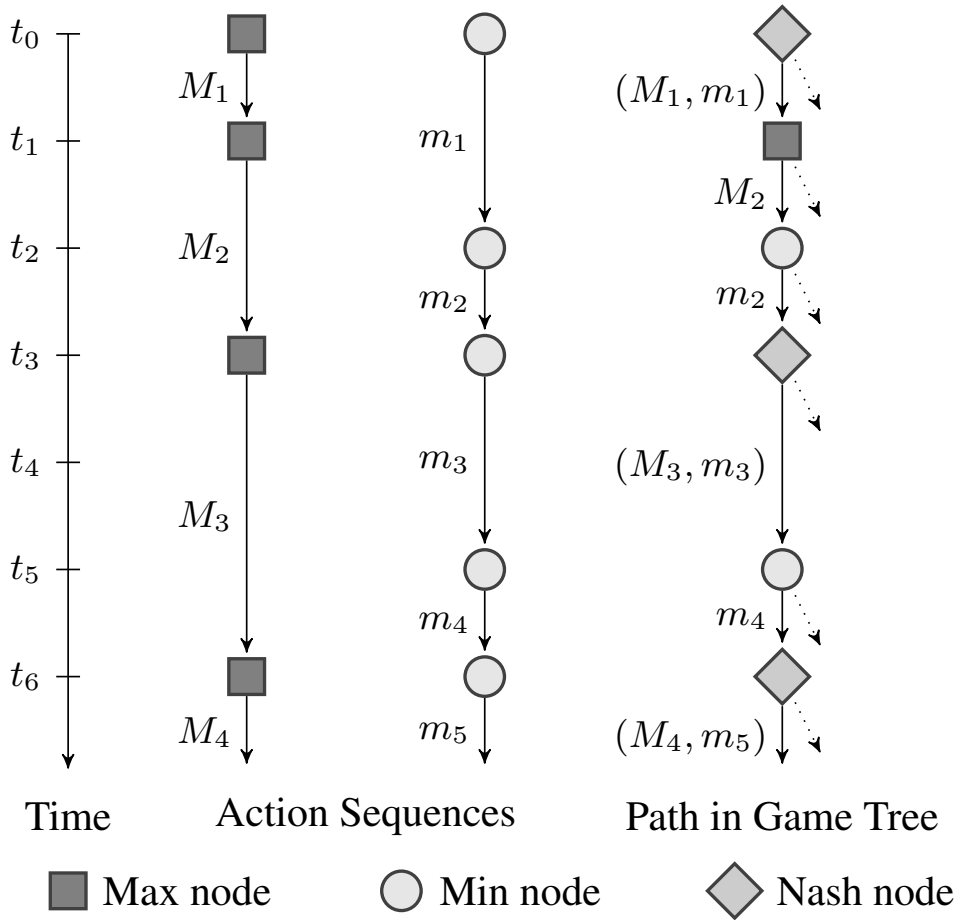


Figure 4.1: Actions with durations. We call a node a *Nash node* when both players can act simultaneously.

to perform an action. Thus, non-trivial decision points for players do not occur on every frame.

Given a player p in a state s, define the next time where p is next able to do a non-pass move by $\tau(s, p) = \min_{u \in s.U_p}(u.\mathrm{t}_a, u.\mathrm{t}_m)$. Note that for any time step t such that $s.\mathrm{t} < t < \min(\tau(s, MAX), \tau(s, MIN))$, players cannot perform any move but pass. It is therefore possible to shortcut many trivial decision points between $s.\mathrm{t}$ and $\min(\tau(s, MAX), \tau(s, MIN))$. Assume an evaluation function has been picked, and remaining simultaneous choices are sequentialized as suggested above. It is then possible to adapt existing search algorithm such as Alpha-Beta or Monte-Carlo Tree Search to take advantage of durative moves.

4.3.2 Evaluation Functions

As with any heurisitc tree search method, we can not hope to search the exponentially large tree in any reasonable time frame, and so we must construct a function for evaluating leaf nodes in our search. A straight-forward evaluation function for combat games is the hitpoint-total differential, i.e.

$$e(s) = \sum_{u \in U_1} \text{hp}(u) - \sum_{u \in U_2} \text{hp}(u)$$

which, however, doesn't take into account other unit properties, such as damage values and cooldown periods. Kovarsky and Buro [47] propose an evaluation based on the life-time damage a unit can inflict, which is proportional to its hp times its damage-per-frame ratio:

$$\text{dpf}(u) = \frac{\text{damage}(w(u))}{\text{cooldown}(w(u))}$$

$$\text{LTD}(s) = \sum_{u \in U_1} \text{hp}(u) \cdot \text{dpf}(u) - \sum_{u \in U_2} \text{hp}(u) \cdot \text{dpf}(u)$$

A second related evaluation function proposed in [47] favours uniform hp distributions:

$$\text{LTD2}(s) = \sum_{u \in U_1} \sqrt{\text{hp}(u)} \cdot \text{dpf}(u) - \sum_{u \in U_2} \sqrt{\text{hp}(u)} \cdot \text{dpf}(u)$$

While these evaluation functions are exact for terminal positions, they can be drastically inaccurate for many non-terminal positions. To improve state evaluation by also taking other unit properties such as speed and weapon range into account, we can try to simulate a game and use the outcome as an estimate of the utility of its starting position. This idea is known as *performing a playout* in game tree search and is actually a fundamental part of Monte Carlo Tree Search (MCTS) algorithms which have revolutionized computer GO [27]. However, there are differences between the playouts we advocate for combat games and previous work on GO and HEX: the playout policies we use here are deterministic. Due to the nature of RTS combat, randomized playouts

on average take far too long to terminate due to the open-world nature of troop movement.

4.3.3 Move Ordering

It is well-known in the game AI research community that a good move ordering fosters the performance of the Alpha-Beta algorithm [70]. When transposition tables (TTs) and iterative deepening are used, reusing previous search results can improve the move ordering. Suppose a position p needs to be searched at depth d and was already searched at depth d'. If $d \leq d'$, the value of the previous search is sufficiently accurate and there is no need for an additional search on p. Otherwise, a deeper search is needed, but we can explore the previously found best move first and hope for more pruning. When no TT information is available, we can use scripted strategies to suggest moves. We call this new heuristic *scripted move ordering*. Note that this heuristic could also be used in standard sequential games like CHESS. We believe the reason it has not been investigated closely in those contexts is the lack of high quality scripted strategies.

Algorithm 3 Alpha-Beta (Considering Durations)

```
1: procedure ABCD(s, d, m_0, α, β)
2:     if computationTime.elapsed then return timeout
3:     else if terminal(s, d) then return eval(s)
4:     toMove ← s.playerToMove(policy)
5:     while m ← s.nextMove(toMove) do
6:         if s.bothCanMove and m_0 = ∅ and d ≠ 1 then
7:             val ←ABCD(s, d − 1, m, α, β)
8:         else
9:             s' ← copy(s)
10:            if m_0 ≠ ∅ then s'.doMove(m_0)
11:            s'.doMove(m)
12:            v ←ABCD(s', d − 1, ∅, α, β)
13:        if toMove = MAX and (v > α) then α ← v
14:        if toMove = MIN and (v < β) then β ← v
15:        if α ≥ β then break
16:    return toMove = MAX ? α : β
```

4.4 Alpha-Beta Considering Durations

In [26] we implemented the proposed combat model, the scripted strategies, the new Alpha-Beta considering durations (ABCD) algorithm, and various tree transformations. We then ran experiments to measure 1) the influence of the suggested search enhancements for determining the best search configuration, and 2) the real-time exploitability of scripted strategies. The scripts used in the experiments are described in sub-section 4.2.1. Note that most of the scripts we described make decisions on an individual unit basis, with some creating the illusion of unit collaboration (by concentrating fire on closest or weakest or most-valuable units). NOK-AV is the only script in our set that exhibits true collaborative behaviour by sharing information about unit targeting. We also tested the following tree transformations: Alt, Alt', and RAB', where Alt' in simultaneous move nodes selects the player that acted last, and RAB' selects the player to move like RAB, but only completes one Alpha-Beta search.

4.4.1 Experiment Setup

The combat scenarios we used for the experiments involved equally sized armies of n versus n units, where n varied from 2 to 8. 1 versus 1 scenarios were omitted due to over 95% of them resulting in draws. Four different army types were constructed to mimic various combat scenarios. These armies were: *Marine* Only, *Marine + Zergling*, *Dragoon + Zealot*, and *Dragoon + Marine*. Armies consisted of all possible combinations of the listed unit type with up to 4 of each, for a maximum army size of 8 units. Each unit in the army was given to player *MAX* at random starting position (x, y) within 256 pixels of the origin, and to player *MIN* at position $(-x, -y)$, which guaranteed symmetric start locations about the origin. Once combat began, units were allowed to move freely in any direction with no boundaries. Unit movement was limited to up, down, left, right at 15 pixel increments, which is equal to the smallest attack range of any unit in our tests. These settings ensured that the matches were fair and symmetric, and would end in a draw of both players played optimally. If the battle did not end in one player being eliminated after 500 actions, the simulation was halted and the final state evaluated with LTD. For instance, in a match between a player p_1 and an opponent p_2, we would count the number of wins by p_1, w, and number of draws, d, over n games and compute $r = (w + d/2)/n$. If both players perform equally, then $r \approx 0.5$.

As the 2011 STARCRAFT AI Competition allowed for 50ms of processing per game logic frame, we gave each search episode a time limit of 5ms. This simulates the real-time nature of RTS combat, while leaving plenty of time (45ms) for other processing which may have been needed for other computations. Experiments were run single-threaded on an Intel Core i7 2.67 GHz CPU with 24 GB of 1600 MHz DDR3 RAM using the Windows 7 64 bit operating system and Visual C++ 2010. A transposition table of 5 million entries (20 bytes each) was used. Due to the depth-first search nature of the algorithm, very little additional memory is required to facilitate search. Each result table entry is the result of playing 365 games, each with random symmetric starting positions.

4.4.2 Influence of the Search Settings

To measure the impact of certain search parameters, we perform experiments using two methods of comparison. The first method plays static scripted opponents vs. ABCD with various settings, which are then compared. The second method plays ABCD vs. ABCD with different settings for each player. We start by studying the influence of the evaluation function selection on the search performance. Preliminary experiments revealed that using NOK-AV for the playouts was significantly better than using any of the other scripted strategies. The playout-based evaluation function will therefore always use the NOK-AV script.

We now present the performance of various settings for the search against script-based opponents (Table 4.1) and search-based opponents (Table 4.2). In Table 4.1, the Alt sequentialization is used among the first three settings which allow to compare the leaf evaluations functions LTD, LTD2, and playout-based. The leaf evaluation based on NOK-AV playouts is used for the last three settings which allow to compare the sequentialization alternatives described in Subsection 4.2.2.

We can see based on the first three settings that performing ABCD with a stronger playout policy evaluation leads to much better performance than with a static evaluation function. ABCD using the NOK-AV playout strategy is indeed dominating the searches using LTD and LTD2 against any opponent tested. We can also see based on the last three settings that the Alt and Alt' sequentializations lead to better results than RAB'.

4.4.3 Estimating the Quality of Scripts

The quality of scripted strategies can be measured in at least two ways: the simplest approach is to run the script against multiple opponents and average the results. To this end, we can use the data presented in Table 4.1 to conclude that NOK-AV is the best script in our set. Alternatively, we can measure the exploitability of scripted strategies by determining the score a theoretically optimal best-response-strategy would achieve against the script. However, such

Table 4.1: ABCD vs. Script - scores for various settings

Opponent	ABCD Search Setting				
	Alt LTD	Alt LTD2	Alt NOK-AV	Alt' Playout	RAB' Playout
Random	0.99	0.98	1.00	1.00	1.00
Kite	0.70	0.79	0.93	0.93	0.92
Kite-AV	0.69	0.81	0.92	0.96	0.92
Closest	0.59	0.85	0.92	0.92	0.93
Weakest	0.41	0.76	0.91	0.91	0.89
AV	0.42	0.76	0.90	0.90	0.91
NOK-AV	0.32	0.64	0.87	0.87	0.82
Average	0.59	0.80	0.92	0.92	0.91

Table 4.2: Playout-based ABCD performance

Opponent	Alt NOK-AV Playout	Alt' NOK-AV Playout	RAB' NOK-AV Playout
Alt-NOK-AV		0.47	0.46
Alt'-NOK-AV	0.53		0.46
RAB'-NOK-AV	0.54	0.54	
Average	0.54	0.51	0.46

Table 4.3: Real-time exploitability of scripted strategies.

Random	Weakest	Closest	AV	Kiter	Kite-AV	NOK-AV
1.00	0.98	0.98	0.98	0.97	0.97	0.95

strategies are hard to compute in general. Looking forward to modelling and exploiting opponents, we would like to approximate best-response strategies quickly, possibly within one game simulation frame. This can be accomplished by replacing one player in ABCD by the script in question and then run ABCD to find approximate best-response moves. The obtained tournament result we call the *real-time exploitability* of the given script. It constitutes a lower bound (in expectation) on the true exploitability and tells us about the risk of being exploited by an adaptive player. Table 4.3 lists the real-time exploitability of various scripted strategies. Again, the NOK-AV strategy prevails, but the high value suggests that there is room for improvement.

4.4.4 Discussuion

In this section we have presented a framework for fast Alpha-Beta search for RTS game combat scenarios of up to 8 vs. 8 units and evaluate it under harsh real-time conditions. This method was based on an efficient combat game abstraction model that captures important RTS game features, including unit motion, an Alpha-Beta search variant (ABCD) that can deal with durative moves and various tree transformations, and a novel way of using scripted strategies for move ordering and depth-first-search state evaluation via playouts. The experimental results are encouraging. Our search, when using only 5 ms per episode, defeats standard AI scripts as well as more advanced scripts that exhibit kiting behaviour and minimize overkill. The prospect of opponent modelling for exploiting scripted opponents is even greater: the practical exploitability results indicate large win margins best-response ABCD can achieve if the opponent executes any of the tested combat scripts.

4.5 UCT Considering Durations

With the success of ABCD, we wanted to explore the possibility of applying Monte-Carlo Tree Search (MCTS) to the same combat model to compare the results to those of ABCD. Our next paper [24] was a follow-up to [26] in which new algorithms were developed tested along with ABCD in order to determine which algorithm performed best in larger combat scenarios of up to 50 vs. 50 units. The UCT algorithm was modified in a similar way to ABCD to create UCT Considering Durations (UCT-CD), shown in Algorithm 4.

There are two main differences between UCT-CD and traditional UCT. The first difference is the modification made to the algorithm to allow for the playing of games which have durative and simultaneous actions, similar to how ABCD was modified. The second difference is that instead of the randomized playouts performed in traditional Monte-Carlo tree search techniques, for UCT-CD we implemented the same deterministic playout policies used in ABCD. Randomized playouts yield poor results in domains such as real-time strategy games due to the complex nature of the game mechanics. Randomized playouts work very well in domains like GO where the game is guaranteed to end even if random moves are played, whereas in RTS games performing randomized moves means units move around the map in random directions rarely attacking each other, meaning the playout takes a very long time to end (if it ever does).

The following section on our new Portfolio Greedy Search algorithm will show experimental results comparing the performance of ABCD, UCT-CD, and Portfolio Greedy Search.

Algorithm 4 UCT Considering Durations

```
1: procedure UCTCD(State s)
2:     root ← new Node
3:     for i ← 1 to maxTraversals do
4:         Traverse(root, Clone(s))
5:         if timeElapsed > timeLimit then break
6:     return most visited move at root
7:
8: procedure TRAVERSE(Node n, State s)
9:     if n.visits = 0 then
10:        UpdateState(n, s, true)
11:        score ← s.eval()
12:    else
13:        UpdateState(n, s, false)
14:        if n.isTerminal() then
15:            score ← s.eval()
16:        else
17:            if !n.hasChildren() then
18:                generateChildren(s, n)
19:            score ← Traverse(SelectNode(n), s)
20:    n.visits++
21:    n.updateTotalScore(score)                    ▷ w.r.t. player to move
22:    return score
23:
24: procedure SELECTNODE(Node n)
25:    bestScore ← −∞
26:    for child c in n.getChildren() do
27:        if c.visits = 0 then return c
28:        score ← (c.totalScore / c.visits) + K · √(log (n.visits)/c.visits)
29:        if score > bestScore then
30:            bestScore ← score
31:            bestNode ← c
32:    return bestNode
33:
34: procedure UPDATESTATE(Node n, State s, bool leaf)
35:    if (n.type ≠ FIRST) or leaf then
36:        if n.type = SECOND then
37:            s.makeMove(n.parent.move)
38:        s.makeMove(n.move)
```

4.6 Portfolio Greedy Search

In [24] we introduced Portfolio Greedy Search (PGS): a new any-time greedy search algorithm for making decisions in complex real-time games with large state and action spaces. Search algorithms such as Alpha-Beta and UCT attempt to search as many actions as possible from a given state in order to cover a large portion of the search space. They then recursively search child nodes deeper into the tree in order to determine which actions at the root will yield beneficial future states. Move-ordering schemes such as those discussed in Subsecion 4.6.2 can be implemented to reduce the branching factor, but it can be that they are still quite large. For RTS combat scenarios, the number of actions possible from any state is the combination of all possible actions by each unit, which is approximately L^U where L is the average number of legal moves per unit, and U is the number of units which can act. Also an issue for traditional search techniques is inaccurate evaluations for non-terminal nodes, which has improved with the introduction of scripted playouts, but still suffers from the fact that these playouts apply a single script policy to every unit in the state. Portfolio Greedy Search deals with these issues in several ways:

- It reduces the number of actions searched for each unit by limiting them to actions produced by a set of scripts called a *portfolio*
- Instead of searching an exponential number of combinations of unit actions, it instead applies a hill-climbing technique to reduce this to a linear amount
- It does not perform any recursive tree search, but instead relies on accurate heuristic evaluations at the root node
- It improves the quality of heuristic evaluation by performing playouts with individually chosen unit-script assignments, rather than assuming all units follow the same policies during the playout.

4.6.1 Algorithm

Portfolio Greedy Search takes as input an initial RTS combat state, a set of scripts to be searched called a *portfolio*, and two integer values I and R. I is the number of improvement iterations we will perform, and R is the number of *responses* we will perform. As output it produces a player move, similar to the output of Alpha-Beta or UCT. The algorithm can be broken down into three main procedures:

- The main procedure PortfolioGreedySearch sets up the initial players and performs the main loops for improving the player policies. Players are initially *seeded* by the GetSeedPlayer procedure that returns an initial player which can then be improved upon via the hill-climbing Improve procedure. After we have improved our player, we can then improve our enemy by the same method, and re-improve our player based on the now stronger opponent. This process is repeated as many times as desired and the resulting player policy is returned.

- The GetSeedPlayer procedure can be seen on line 14. This procedure produces an initial policy to be implemented by all units the player controls. To do this, it iterates over all scripts in our portfolio, setting each unit's policy to the current script, and then perform a playout with each iteration. We then set our player's initial seed policy to the best performing script found via this process.

- The Improve procedure is the most important part of the Portfolio Greedy Search algorithm. Instead of searching an exponentially large combination of all possible unit actions, it instead uses a hill-climbing procedure to search over each script in our portfolio exactly once for each unit. At each iteration it performs a playout using the individual unit-script assignments, the result is recorded, and after each script has been applied to a unit, that unit's script is set to the best one found so far during the process.

Algorithm 5 Portfolio Greedy Search

```
1: Portfolio P                                          ▷ Script Portfolio
2: Integer I                                            ▷ Improvement Iterations
3: Integer R                                            ▷ Self/Enemy Improvement Responses
4: Script D                                             ▷ Default Script
5:
6: procedure PortfolioGreedySearch(State s, Player p)
7:     Script enemy[s.numUnits(opponent(p))].fill(D)
8:     Script self[] ← GetSeedPlayer(s, p, enemy)
9:     enemy ← GetSeedPlayer(s, opponent(p), self)
10:    self = Improve(s, p, self, enemy)
11:    for r = 1 to R do
12:        enemy = Improve(s, opponent(p), enemy, self)
13:        self = Improve(s, p, self, enemy)
14:    return generateMoves(self)
15:
16: procedure GetSeedPlayer(State s, Player p, Script e[])
17:    Script self[s.numUnits(p)]
18:    bestValue ← −∞
19:    Script bestScript ← ∅
20:    for Script c in P do
21:        self.fill(c)
22:        value ← Playout(s, p, self, e)
23:        if value > bestValue then
24:            bestValue ← value
25:            bestScript ← c
26:    self.fill(bestScript)
27:    return self
28:
29: procedure Improve(State s, Player p, Script self[], Script e[])
30:    for i = 1 to I do
31:        for u = 1 to self.length do
32:            if timeElapsed > timeLimit then return
33:            bestValue ← −∞
34:            Script bestScript ← ∅
35:            for Script c in P do
36:                self[u] ← c
37:                value ← Playout(s, p, self, e)
38:                if value > bestValue then
39:                    bestValue ← value
40:                    bestScript ← c
41:            self[u] ← bestScript
42:    return self
```

4.6.2 Experiments

Two main sets of experiments were carried out to compare the performance of ABCD, UCTCD, and the new Portfolio Greedy Search algorithms. The first set of experiments play ABCD vs. UCTCD, in order to show the comparative strength of the two baseline search algorithms. The second set of experiments then play ABCD and UCT vs. the proposed Portfolio Greedy Search algorithm to see how it performs against the current state of the art.

Combat Scenario Setup

Each experiment consists of a series of combat scenarios in which each player controls an identical group of n STARCRAFT units. To show how each algorithm performs in large combat scenarios, each experiment was repeated for values of n equal to 8, 16, 32, and 50, 50 being roughly the size of the largest battles seen in a typical game of STARCRAFT. Further, two different geometric configurations of the initial unit states were used:

- **Symmetric** states, in which units for each player are placed randomly symmetric about the midpoint m of the battlefield. For each unit in position $m+(x, y)$ for player 1, player 2 receives the same unit at position $m+(-x, -y)$. This ensures a fair initial starting position, but one which would not typically be seen in an RTS combat setting.

- **Separated** states were designed to more closely resemble an actual RTS combat scenario. A midpoint m for the battlefield is chosen, and then each player's force is generated randomly symmetric to the midpoint, and then translated a fixed distance d to the left or right. For example, a unit for player 1 generates a random (x, y) position and is placed at location $m + (x - d, y)$ with player 2's identical unit being placed at position $m + (-x + d, -y)$. Distance d was chosen so that it is larger than the largest attack radius of any unit, so that both groups of unit are separated before attacking begins, simulating two opposing forces clashing on a battlefield. Each separated state is generated twice, with each force appearing once on the left and once on the right, for fairness.

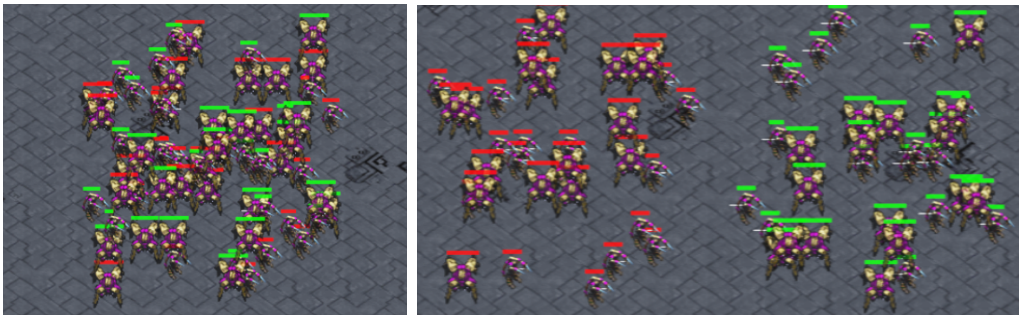


Figure 4.2: A *symmetric* state (left) and a *separated* state (right).

For both symmetric and separated states, random positions (x, y) were generated with bounds of $x, y \in [-128, 128]$ pixels. This kept a decent spacing of starting units, while mimicking the tight formation of a typical group of units in a combat scenario. The battlefield itself was an enclosed arena with width 1280 pixels and height 720 pixels, with midpoint position $m = (640, 360)$. Units were free to move anywhere within the arena, but could not move through the "walls" at the outer edges. An enclosed arena was used to ensure that each battle eventually terminated, as an infinite plane resulted in many cases of one player simply running away from a fight indefinitely.

Although movement in SparCraft can be performed in any direction, for our experiments we limit movement to only allow fixed length movements up, down, left, or right. This abstraction is necessary to reduce the search space for each algorithm. Although this abstraction may seem quite coarse, by setting a small movement length of 8 pixels the movement of units in the simulator appears quite similar to the actual game of STARCRAFT. For each set of experiments, 5 different different configurations of starting unit types were also used to simulate various RTS army compositions with both melee and ranged units of different strengths. Also, early game units were used as they are by far the most commonly seen units in STARCRAFT combat. The following were used as starting unit type counts for each player for each battle of size n units:

- n Protoss Dragoons (Strong Ranged)
- n Zerg Zerglings (Weak Melee)

- $n/2$ Protoss Dragoons with $n/2$ Protoss Zealots (Strong Melee)
- $n/2$ Protoss Dragoons with $n/2$ Terran Marines (Weak Ranged)
- $n/2$ Terran Marines with $n/2$ Zerg Zerglings

100 randomly generated battles were carried out for each of the 5 starting unit configurations, giving 500 total battles for each separated state and for each symmetric state experiment for each tested value of n starting units.

Environment and Search Settings

All experiments were performed on an Intel(R) Core(TM) i7-3770K CPU @ 3.50GHz running Windows 7 Professional Edition, with all algorithms running single-threaded. A total of 12 GB DDR3 1600MHz RAM was available, however the maximum amount of RAM consumed by any process monitored at less than 14 MB, which was used to store both the UCT search tree and the Alpha-Beta transposition table. Experiments were programmed in C++ and compiled using Visual Studio 2012.

Search Algorithm Parameters

Each search algorithm was given a 40 ms time limit per search episode to return a move at a given state. This time limit was chosen to mimic real-time performance in STARCRAFT, which runs at 24 fps (42 ms per frame). Alpha-Beta and UCT search algorithms were given an upper limit of 20 children per search node. Due to the exponential number of possible actions at each search state, having no upper bound on the number of children at a node would often produce searches which did not leave the root node of a tree, which produced very bad results. In practice we found that imposing a child limit, when combined with clever move-ordering (next section) produce best results.

- Alpha-Beta search:
  - Time Limit: 40 ms
  - Max Children: 20
  - Evaluation: NOK-AV vs. NOK-AV Playout
  - Transposition Table Size: 100000 (13.2 MB)
- UCT search:
  - Time Limit: 40 ms
  - Max Children: 20
  - Evaluation: NOK-AV vs. NOK-AV Playout
  - Final Move Selection: Most Visited
  - Exploration Constant: 1.6
  - Child Generation: One-at-leaf
  - Tree Size: No Limit (6 MB largest seen in 40 ms)
- Portfolio Greedy search:
  - Time Limit: 40 ms
  - Improvement Iterations I: 1
  - Response Iterations R: 0
  - Initial Enemy Script: NOK-AV
  - Evaluation: Improved Playout
  - Portfolio Used: (NOK-AV, Kiter)

Of note is the choice of low settings for $I = 1$ and $R = 0$. These were chosen for two reasons: first, to show the performance of the base settings for Portfolio Greedy Search, and also because higher settings do not yet run within 40 ms.

Move Ordering

It is well known that with game tree search algorithms such as Alpha-Beta or UCT, a good move-ordering scheme can greatly improve performance [70]. If better moves are searched first, Alpha-Beta can produce better cuts and search deeper, while if UCT searches better nodes first, it will spend less time exploring less valuable moves. With a child limit imposed on our search, we must ensure that the moves we search are useful, and we do this in several ways. At each search node, Alpha-Beta and UCT first search the moves generated by our NOK-AV and Kiter scripts. These moves are then followed by moves containing Attack actions, then by moves containing Movement actions. Movement actions are explored in random order for fairness. Also, Alpha-Beta first considers moves which have been stored in the transposition table.

Opponent Modelling

Experiments involving Alpha-Beta (AB) and UCTCD (UCT) were conducted with two opponent-modelling parameter settings: either all opponent actions were searched in the game tree, or opponent actions were fixed to that of the NOK-AV script. By fixing the enemy actions, we are effectively approximating a best response to that script in an attempt to exploit it. This was shown to give a substantial performance gain against scripted opponents in [26], and so we tested to see if it would have any effect against Portfolio Greedy Search, which searches over scripted moves.

4.6.3 Results

Parameter optimization was performed on the exploration constant K of the UCT algorithm (Algorithm 1, line 29) to ensure good performance in our experiments. The results from this optimization can be seen in Fig. 4.4, which determined that the choice of constant did not highly affect results in either the symmetric or separated state experiments against Alpha-Beta. We chose a value of 1.6, which was the value with the highest result sum from both experiments.

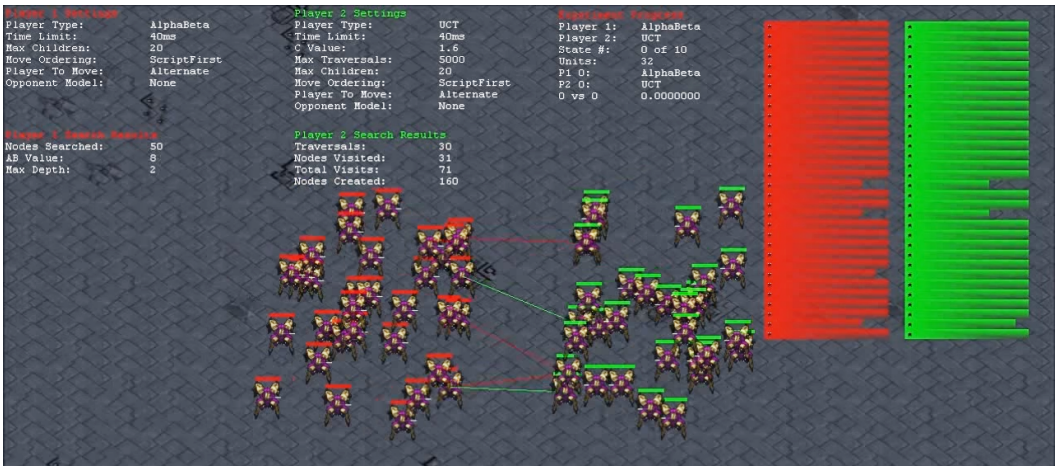
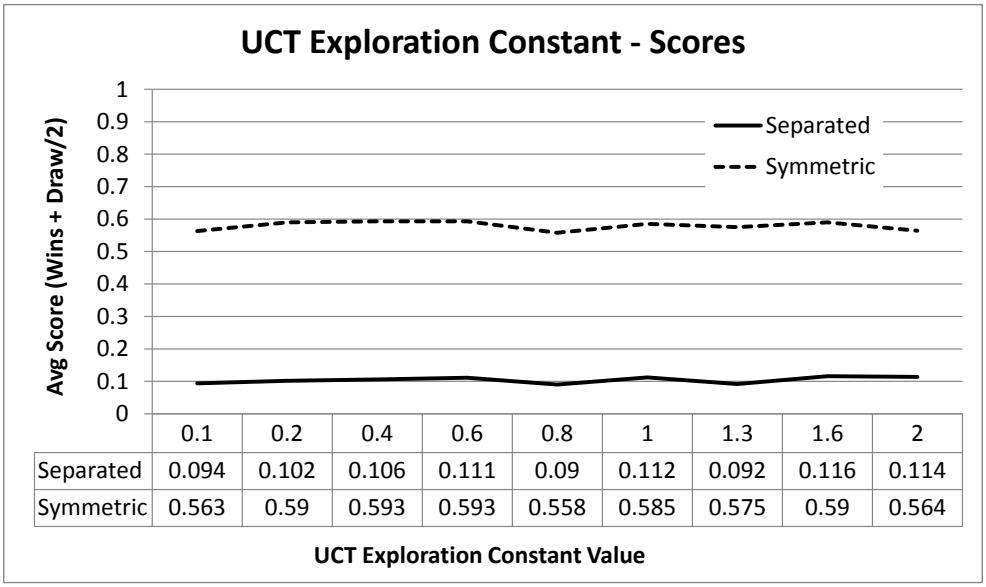


Figure 4.3: A screenshot of the SparCraft combat visualization system with a scenario consisting of 32 vs. 32 Protoss Dragoons. The left player is being controlled by ABCD and the the right player is being controlled by UCT-CD.



	0.1	0.2	0.4	0.6	0.8	1	1.3	1.6	2
Separated	0.094	0.102	0.106	0.111	0.09	0.112	0.092	0.116	0.114
Symmetric	0.563	0.59	0.593	0.593	0.558	0.585	0.575	0.59	0.564

Figure 4.4: Average scores for various settings of UCT exploration constant K. Experiments were performed vs. Portfolio Greedy Search with 8, 16, 32, and 50 starting units for both separated and symmetric states. $K = 1.6$ was chosen for the paper's main experiments.

Search vs. Script

Experiments were performed with Alpha-Beta, UCT, and Portfolio Greedy Search against each script type listed in 4.2.1, with all 3 search techniques achieving a win rate of 100% against scripted players for all battle sizes.

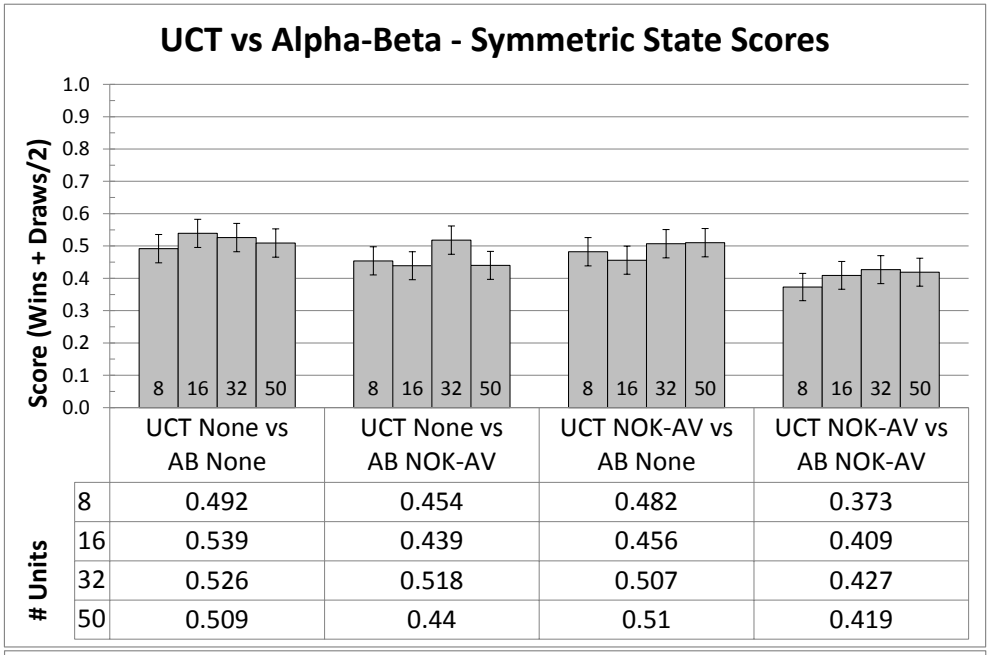
UCT vs. Alpha-Beta

The results from the UCT vs. Alpha-Beta experiment can be seen in Fig. 4.5. Immediately one notices the dramatic difference in the result between symmetric state and separated state types. Experiments performed in symmetric states tend to show equal performance between both algorithms, except for the case where both UCT and Alpha-Beta are configured to compute a best response to the NOK-AV script. Experiments on separated states (the more realistic of the two types) show that for small battles, both methods perform equally well, but UCT outperforms Alpha-Beta as the battles grow larger.

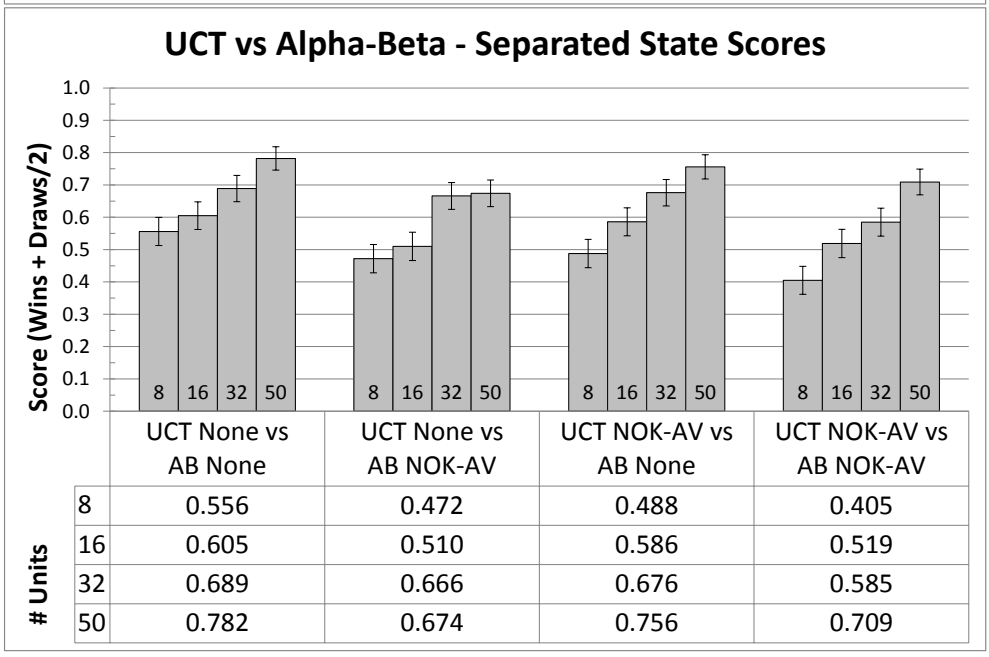
A possible explanation for the difference in results between the two state types is intuitive: in symmetric states, units are usually within firing range of many other units, and since there is a small reload-speed penalty for moving (as is present in STARCRAFT), the problem reduces almost entirely to a unit-targeting problem. By almost completely eliminating the need for clever movement, neither search algorithm can gain an advantage over the other through search. For separated states, there is much more room for clever tactics such as kiting, retreating when at low health, group formations, etc. Since both search algorithms are given identical action spaces to search, this shows that the UCT algorithm is better suited for larger RTS combat scenarios than Alpha-Beta.

Portfolio Greedy Search

Results from the Portfolio Greedy Search algorithm can be seen in Fig. 4.6. As in the previous experiment, the results for symmetric states are fairly even, with the exception versus the Alpha-Beta algorithm which computes a best response to NOK-AV. Because NOK-AV is one of the two scripts in the portfolio



# Units	UCT None vs AB None	UCT None vs AB NOK-AV	UCT NOK-AV vs AB None	UCT NOK-AV vs AB NOK-AV
8	0.492	0.454	0.482	0.373
16	0.539	0.439	0.456	0.409
32	0.526	0.518	0.507	0.427
50	0.509	0.44	0.51	0.419

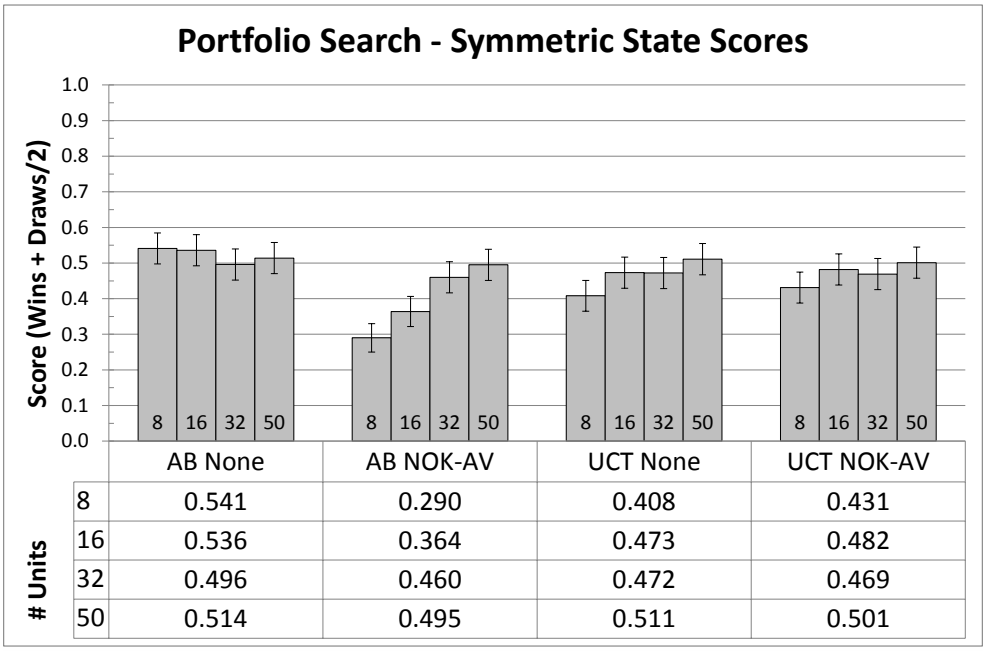


# Units	UCT None vs AB None	UCT None vs AB NOK-AV	UCT NOK-AV vs AB None	UCT NOK-AV vs AB NOK-AV
8	0.556	0.472	0.488	0.405
16	0.605	0.510	0.586	0.519
32	0.689	0.666	0.676	0.585
50	0.782	0.674	0.756	0.709

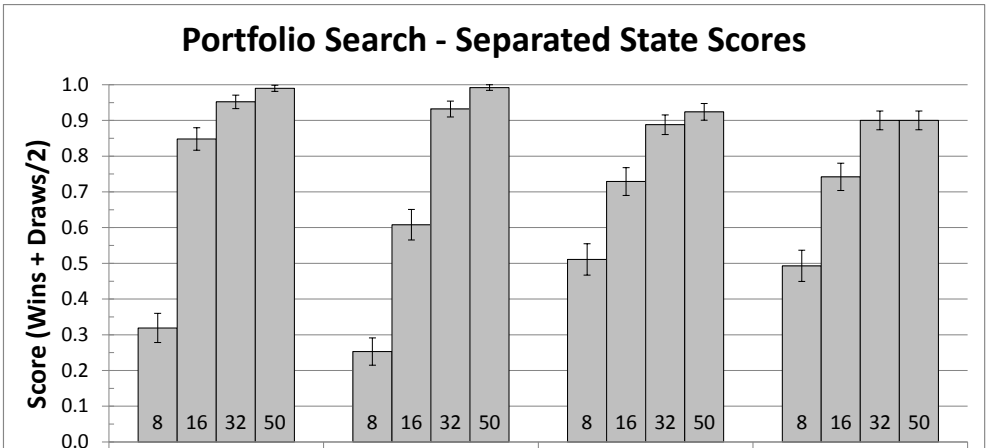
Figure 4.5: Results of Alpha-Beta vs. UCT for Symmetric States (top) and Separated States (bottom). Both algorithms have two configurations, one without opponent modelling labelled "None", and with modelling against script NOK-AV. Results are shown for combat scenarios of n vs. n units, where $n = 8, 16, 32, 50$. 500 combat scenarios were played out for each configuration. 95% confidence error bars are shown for each experiment.

and symmetric states tend to favour no movement, NOK-AV will be the script chosen by the greedy search the majority of the time. As shown in [26], this type of best response computation can be quite powerful in exploiting scripted behaviours. However, these results also show that UCT does far worse than Alpha-Beta at performing this exploitation.

The separated state results show that the portfolio greedy search algorithm easily defeats Alpha-Beta and UCT for larger state sizes. While performance is weak for 8 vs. 8 units, as combat scenarios increase in size it dominates the traditional search algorithms, winning nearly all battles against Alpha-Beta and more than 90% of battles against UCT. Fig. 4.7 shows average execution times of complete Portfolio Greedy Search search episodes with respect to the number of units in a separated state scenario, if no time limit had been specified. This graph illustrates the quick running time of the Portfolio Greedy Search algorithm with respect to traditional tree search methods which would require vast computational resources to fully search large scenarios. We can see that the time limit of 40 ms was only reached when performing searches on states with more than 2×25 units. Of note is the quadratic running time with respect to the number of units in the scenario, which one would expect to be linear due to nature of the algorithm. This is explained by the use of playouts for state evaluations whose running times are themselves linear with respect to the number of units in a scenario, due to the need for an action to be calculated for each unit. Execution times were recorded only for the first move of symmetric and separated states in order to illustrate their differences, which exist due to the underlying scripts in the portfolio. Since the scripts are optimized to choose attack actions before move actions, they encounter their worst-case running time on initially separated states in which no attack options are found, forcing all move options to be explored. However, once both opposing forces of a separated state engage in battle, their values approach that of symmetric states (on average for the duration of the battle).



# Units	AB None	AB NOK-AV	UCT None	UCT NOK-AV
8	0.541	0.290	0.408	0.431
16	0.536	0.364	0.473	0.482
32	0.496	0.460	0.472	0.469
50	0.514	0.495	0.511	0.501



# Units	AB None	AB NOK	UCT None	UCT NOK
8	0.319	0.253	0.511	0.493
16	0.848	0.608	0.729	0.742
32	0.952	0.932	0.888	0.900
50	0.990	0.992	0.924	0.900

Figure 4.6: Results of Portfolio Greedy Search vs. Alpha-Beta and UCT for Symmetric States (top) and Separated States (bottom). Both algorithms have two configurations, one without opponent modelling labelled "None", and with modelling against script NOK-AV. Results are shown for combat scenarios of n vs. n units, where $n = 8, 16, 32, 50$. 500 combat scenarios were played out for each configuration. 95% confidence error bars are shown for each experiment.

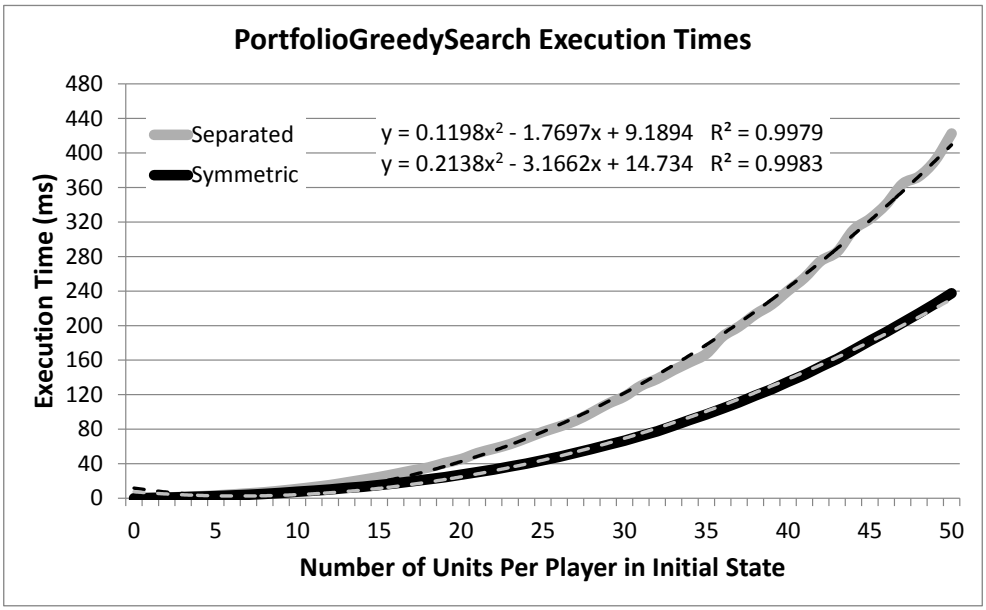


Figure 4.7: Graph showing average execution times of complete Portfolio Greedy Search search episodes with respect to the number of units in the combat scenario when no time limit is specified. Execution times are extracted from the first move from the initial symmetric or separated states. Sample standard deviations for symmetric state running times for different unit numbers are: 10 units: 2.3 ms, 25 units: 9.0 ms, 50 units: 55.5 ms, and for separated states: 10 units: 2.2 ms, 25 units: 19.7 ms, 50 units: 111.5 ms.

4.6.4 Discussion

In sections 4.5 and 4.6 we presented a modified version of UCT for handling games with simultaneous and durative actions, as well as a new greedy search algorithm for RTS combat: Portfolio Greedy Search. We have implemented and shown experimental results comparing Alpha-Beta, UCT, Portfolio Greedy Search, for use in RTS game combat scenarios. We have shown that UCT outperforms Alpha-Beta in battle scenarios with realistic unit positions (separated states) as battle sizes get larger. We have also shown that the new Portfolio Greedy Search algorithm outperforms both Alpha-Beta and UCT for medium to large size separated state battle scenarios, winning over 90% of battles with more than 32 units. This new Portfolio Greedy Search algorithm is currently the state of the art in large-scale real-time strategy game combat.

Several improvements can be made to the Portfolio Greedy Search algorithm which can improve both its speed and results. Using portfolio P, Portfolio Greedy Search performs $|P|$ playouts per unit per search. These playouts could be trivially parallelised, allowing a linear speed-up in running time with respect to $|P|$. In our case, using a two script portfolio, this would yield a 100% speed increase in the algorithm. To improve performance of Portfolio Greedy Search, extra decision points (iterating over scripts for each unit) could be created in the search tree to improve the accuracy of the evaluation. Unlike tree search methods, Portfolio Greedy Search only optimizes decision at the root node before performing its playout evaluation. By implementing a scheme in which extra search is performed after a certain number of moves have been performed in the playout, it could improve performance. We can then imagine a hybrid tree search algorithm in which Portfolio Greedy Search is the method used by a minimax type algorithm to choose which moves to play at a given node in the tree. Portfolio Greedy Search would need to be significantly faster in order to be used within another tree search algorithm in real-time.

It is our intention to use Portfolio Greedy Search for combat decision making in a future version of , our entry to the STARCRAFT AI Competition. By examining the results in Fig. 4.6, we can see that while Portfolio Greedy Search

performs quite well for larger combat scenarios, it is beaten by Alpha-Beta for smaller scenarios. We can now envision a hybrid AI agent which dynamically chooses which search method to use based on size of the combat scenario presented. Because each algorithm has its strengths and weaknesses, creating an agent which is able to capitalize on all of the strengths with none of the weaknesses seems like the most intelligent choice for future competitions.

4.7 Integration Into RTS AI Agents

So far in this chapter we have discussed various algorithms for decision making in RTS combat scenarios, all of which have used the SparCraft combat simulation system as the experimental test-bed. The question now remains of how well these systems perform in the real game of STARCRAFT Ẇith SparCraft, we have the entire source code of the combat simulation engine, with pixel and frame perfect accuracy in the movements and actions of all units. Unfortunately, the source code of STARCRAFT is not available, and so we must interact with it using an external API, namely BWAPI. In this section we will address the problems related to incorporating search techniques into the STARCRAFT engine through our STARCRAFT AI competition entry: UAlbertaBot (see section 6.1 for details). Please note that the research conducted in this section was performed after the development of the ABCD algorithm, but before the development of the UCT-CD and PGS algorithms.

Despite BWAPI's comprehensive interface into the STARCRAFT game engine, there are still some intuitively simple tasks which require non-trivial effort to implement. Take for example the case of issuing an attack command to a unit in the game. To carry out frame-perfect unit micro-management we will require knowledge of the exact frame in which the unit has fired its weapon and dealt its damage. This is important because STARCRAFT's game engine will cancel an attack command if another command is given before damage has been dealt, resulting in less damage being done by the unit over time. Currently, there is no functionality in BWAPI which can give us this exact information, so it must be extracted via a combination of reverse-engineered

Attack Sequence	isAtk	atkFrm	Additional Notes
1. Unit is Idle	`False`	`False`	Unit currently idle
2. Issue Attack Cmd	`False`	`False`	Player gives attack order
3. Turn to Face Target	`False`	`False`	0 duration if facing
4. Approach Target	`False`	`False`	0 duration if in range
5. Stop Moving	`False`	`False`	Some units stop before firing
6. Begin Attack Anim	`True`	`True`	Attack animation, no dmg yet
7. Anim Until Damage	`True`	`True`	Animation frames until projectile
8. Mandatory Anim	`True`	`True`	Extra animation after damage
9. Optional Anim	`True`	`True`	Other animations such as reload
10. Wait for Reload	`True`	`False`	Unit may move before next atk
11. Goto Step 3	`False`	`False`	Repeat the attack

Table 4.4: Sequence of events occurring after an attack command has been given in StarCraft. Also listed are the associated values of isAtk and atkFrm, the results of BWAPI unit.isAttacking() and unit.isAttackFrame() return values for the given step. This shows the non-triviality of something as intuitively simple of having frame-perfect control of unit actions in STARCRAFT.

game logic and animation script data obtained via a resource extraction program called PyICE.

BWAPI gives us access to two separate functions to help determine if a unit is currently attacking: unit.isAttacking(), which returns true if the unit is currently firing at a unit with intent to continue firing, and unit.isAttackFrame(), which returns true if the unit is current animating with an attack animation frame. Table 4.4 shows the sequence of events which take place after issuing a unit.attack() command in STARCRAFT. Steps 1-5 deal with the unit moving into a position and heading at which it can fire, steps 6-9 deal with the actual firing time of the unit, and step 10 is a period of time where the unit is waiting until it can fire again. This sequence shows that neither function gives us the exact time when the unit dealt its damage, due to steps 8 and 9, which are steps in which these functions return true, but after damage has already been inflicted. We must therefore attempt to extract a more accurate estimate of this information from the STARCRAFT animation data files using PyICE, helping to determine the frame when damage has been dealt (the end of step 7). For a given unit, we extract the duration of steps 6-9 from PyICE and call this value $atkFrames$.

To determine this timing, we will keep track of the unit after we have given an attack command to make sure no other commands are given before the end of step 7. We record the first frame after the attack command was given for which the unit.isAttackFrame() returns true (the beginning of step 6), and call this value *startAtk*. We then calculate the frame in the future when the unit will have dealt its damage by:

$$damageFrame = startAtk + atkFrames$$

By issuing subsequent commands to the unit only after *damageFrame* we hope that no attacks will be interrupted, while allowing the unit to perform other commands between attacks for as long as possible. For example, our data extraction shows that a Protoss Dragoon unit has an attack cooldown of 23 frames, but an *atkFrames* value of 7, which means it has 16 frames after firing that it is free to move around before it fires again, which can be useful for strategic attack sequences such as *kiting*, a technique used against units with short range weapons to avoid taking damage by fleeing outside of its weapon range while waiting to reload. However, despite this effort which should work in theory, in practice the STARCRAFT engine does not behave in a strict deterministic fashion, and work is still being done to perfect this model so that a higher level of precise combat unit control can be obtained.

4.7.1 StarCraft Experiments

To evaluate our combat search AI system in STARCRAFT, we implemented a simplified version of UAlbertaBot in BWAPI which only performs combat scenarios. We allocate only 5 ms per frame to our AI's search algorithm in order to simulate the competition environment in which the full bot is executing each frame. UAlbertaBot's micro-management system involves a policy of "Attack Weakest Enemy Unit in Range", with an option for game commander to retreat the squad from combat if the SparCraft simulation predicts defeat (see Fig. 6.2). For this experiment no retreat was allowed — combat is performed until one team has been eliminated or a time limit of 5000 frames (208 seconds) is reached. In this experiment we construct several STARCRAFT test

maps which contain pre-positioned combat scenarios. To evaluate our combat search system we will do a comparison of its performance to that of the micro-management system present in UAlbertaBot (AttackWeakest). Due to the desire to avoid issues with network latency (necessary to play one combat policy against the other directly) we instead chose to perform combat with both methods vs. the default STARCRAFT AI, and then compare the obtained scores. The default STARCRAFT AI's combat policy is not explicitly known, however it is thought to be approximately equal to the AttackWeakest script, however it does not appear to be fully deterministic. We will then compare the results our BWAPI experiment with results obtained from performing the exact same scenario with the SparCraft simulator . Games were played against the AttackWeakest scripted policy, which is the closest known to that of the default STARCRAFT AI.

The scenarios we construct are designed to be realistic early-mid game combat scenarios which could be found in an actual STARCRAFT game. They have also been designed specifically to showcase a variety of scenarios for which no single scripted combat policy can perform well under all cases, and can be seen in Figure 4.8. Units for each player are shown separated by a dotted line, with the default AI units placed to the right of this line. Unit positions were fixed to the formations shown at the start of each trial, but units were allowed to freely move about the map if they are instructed to do so. For each method, 200 combat trials were performed in each of the scenarios.

Scenario A is designed such that the quicker, ranged Vulture units start within firing range of the zealots, and must adopt a kiting strategy to defeat the slower but stronger melee Zealot units. Scenario B is similar to A. However, two strong ranged Dragoons must also kite a swarm of weaker melee Zerglings to survive. Scenario C is symmetric, with initial positions allowing the Dragoons to reach the opponent zealots, but not the opponent Dragoons. Scenario D is also symmetric, with each unit within firing range of each other unit. Therefore, a good targeting policy will perform well. In addition to these scenarios, four more scenarios A', B', C', and D' were tested, each having a similar formation to the previously listed scenarios. However, their positions

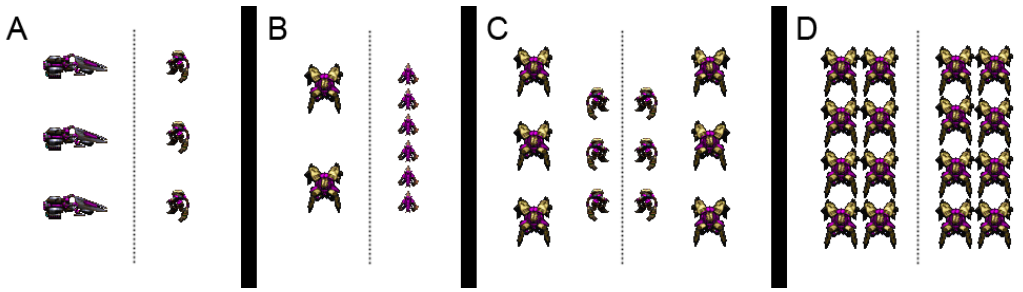


Figure 4.8: Micro search experiment scenarios. A) 3 ranged Vultures vs. 3 melee Zealot. B) 2 ranged Dragoons vs. 6 fast melee Zerglings. C) 3 Dragoon + 3 Zealots in symmetric formation. D) 8 Dragoons in symmetric two-column formation.

are perturbed slightly to break their perfect line formations. In the case of C and D, symmetry was maintained for fairness. These experiments were performed on hardware similar to the build-order planning hardware, with 1 MB total memory used for the search routine and 2 MB for the transposition table.

The results from the micro search experiment are presented in Table 4.5. Shown are scores for a given combat method, which are defined as: $score = wins + draws/2$. We can see from these results that it is possible (through expert knowledge) to design a scripted combat policy (such as Kiter) which will perform well in scenarios where it is beneficial to move out of shorter enemy attack range like in scenarios A/B, but will fail in scenarios where excess movement is detrimental as it imposes an small delay on firing like in scenarios C/D. Scripts such as AttackWeakest perform better than Kiter in scenarios in which its better targeting policy and lack of movement allow for more effect damage output, but fail completely in situations such as A/B where standing still in range of powerful melee enemies spells certain death. By implementing a dynamic AI solution for combat micro problems, we have dramatically improved overall performance over a wide range of scenarios, even while under the extremely small time constraint of 5 ms per frame.

Also of note in these results is the fact that although the scripted strategies are deterministic, the outcome in the actual BWAPI implementation was not always the same for each trial. In a true deterministic and controllable RTS game model (such as our simulator), each of the scripted results should either

	Combat Decision Settings					
	Search (5 ms)		AtkWeakest		Kiter	
	Sim	Game	Sim	Game	Sim	Game
A	1.00	0.81	0	0	1.00	0.99
A'	1.00	0.78	0	0	1.00	0.99
B	1.00	0.65	0	0	1.00	0.94
B'	1.00	0.68	0	0	1.00	0.89
C	1.00	0.95	0.50	0.56	0	0.14
C'	1.00	0.94	0.50	0.61	0	0.09
D	1.00	0.96	0.50	0.58	0	0.11
D'	1.00	0.97	0.50	0.55	0	0.08
Avg	1.00	0.84	0.25	0.29	0.50	0.53

Table 4.5: Results from the micro AI experiment. Shown are scores for Micro Search, AttackWeakest, and Kiter decision policies each versus the built-in STARCRAFT AI for each scenario. Scores are shown for both the micro simulator (Sim) and the actual BWAPI-based implementation (Game).

be all wins, losses, or draws. This surprising result must be due to the nature of the STARCRAFT engine itself, for which we do not have an exact model. It is known that the STARCRAFT engine does have a small level of stochastic behaviour both in its unit hit chance mechanism and its random starting unit heading direction. It is unknown whether or not the default combat policy contains non-deterministic elements. It also highlights an additional frustration of implementing an RTS game bot in a real-world scenario: that results may not always be exactly repeatable, so robust designs are necessary.

In this section we showed how even though we may have developed state-of-the-art algorithms for performing RTS combat search, integrating those algorithms into an actual retail game engine can still be problematic. While the results obtained from the experiments performed using BWAPI in STARCRAFT are promising, they do show that more work is required before our simulator is able to match that of the actual STARCRAFT engine. Similarly, more work is required in finding out the fine-grained details of exactly how the movements and actions of units are implemented in STARCRAFT so that we can truly see the power of search shine in a competition setting. Schneider and Buro

[71] have since performed additional analysis of the details of unit motion in STARCRAFT, which showed that the lack of acceleration in SparCraft led to a divergence of the STARCRAFT and SparCraft game states, however these results have not yet been incorporated back into SparCraft. As such, UAlbertaBot does not yet implement the search-based combat algorithms in competition settings, however it does make extensive use of the SparCraft combat simulator for battle outcome prediction.

Chapter 5

Hierarchical Portfolio Search and the Prismata AI

Most of the focus of the research presented so far in this thesis, and academic game AI literature in general is focused on creating the strongest AI agents possible for a given category of games. When creating commercial video games however, the goal is not limited to maximizing the playing strength of the AI system, but to provide the most enjoyable experience for its users. With high development costs, industry game AI programmers look for ways to automate decision making beyond relying solely on manually tuned behavior, creating AI systems that are more robust to game design changes, and also making them better adjust to human players' preferences and playing strength.

In this chapter we introduce a generalized search procedure for games with large state and action spaces: Hierarchical Portfolio Search (HPS). We discuss HPS's role in creating a strong, robust, and modular AI system for the commercial strategy game Prismata by Lunarch Studios [49]. After discussing specific game AI challenges, we present our new generic search procedure, introduce Prismata, and show game strength evaluations and the results of an AI user survey. This chapter is based on our publication [25] which won the Best Student Paper award at the 2015 Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE) and was selected as an invited talk for the AI Summit of Game Developer's Conference (GDC) 2016.

5.1 AI Design Goals

In order to create the most enjoyable experience for their users, several design goals must be considered when creating modern video game AI systems:

- **New Player Tutorial**: Because new games may have fairly steep learning curves, an AI system should be a tool which aids new players in learning the game rules and strategies. It should also offer different difficulty settings so that players have a gradual introduction rather than being placed immediately at the highest difficulty.

- **Experienced Player Training**: Experienced and competitive players often want to practice without "giving away" strategies to other players. The hardest AI difficulty should be able to put up enough fight so that players can practice these strategies with some resistance.

- **Single Player Replayability**: Single-player missions in video games are usually implemented as scripted sequences of events that play out the same way every time, allowing a player memorize strategies in order to defeat them. In order to add replay value the AI system should be more dynamic, ensuring the player doesn't fight against the same tactics every time they play.

- **Robust to Change**: Unlike traditional board games whose rules remain the same over centuries, the game objects in modern games may have properties that need to be tweaked over time for strategic balancing. If the AI system were based on hard-coded scripts it could require maintenance every time an object was updated, costing valuable time for programmers.

5.2 Hierarchical Portfolio Search

The algorithm we propose for making decisions in large search spaces is called Hierarchical Portfolio Search (HPS), which is based on the portfolio greedy search algorithm described in chapter 4. The key idea of portfolio based search

methods is that instead of iterating over all possible actions for a given state we use a portfolio of algorithms to generate a much smaller, yet (hopefully) intelligent set of actions. This method is particularly useful in scenarios where a player's decision can be decomposed into many individual actions, such as real-time strategy games like StarCraft or collectible card games like Hearthstone or Magic: the Gathering. Typically these decompositions are inspired by *tactical* components of the game such as economy, defense, and offense.

Here we extend the previous methods by creating HPS: a bottom-up, two level hierarchical search system inspired by military hierarchical command structure [90]. At the bottom layer there is a portfolio of algorithms which generate multiple suggestions for each of several tactically decomposed areas of the game turn. At the top layer, all possible combinations of these suggestions are iterated over by a high-level search technique (such as MiniMax or Monte-Carlo tree search) which makes the final decision on which move to perform. While it is possible that this abstraction may not generate the strategically optimal move for a given turn, there may have been so many possible actions for that turn that finding the optimal move was intractable.

5.2.1 Components of HPS

Let us now define the components of the HPS system:

State s containing all relevant game state information

Move $m = \langle a_1, \ldots, a_k \rangle$, a sequence of **Actions** a_i

Player function p $[m = p(s)]$

- Input state s
- Performs Move decision logic
- Returns move m generated by p at state s

Game function g $[s' = g(s, p_1, p_2)]$

- Initial state s and Players p_1, p_2

- Performs game rules / logic
- Returns final game state s' (win, lose, or draw)

These are the basic components needed for most AI systems which work on abstract games. In order to implement Hierarchical Portfolio Search we will need to add two more components to this list. The first is a *Partial Player* function, which like a Player function computes move decision logic, but a Partial Player computes only a partial move for a turn. An example of a partial move would be in an RTS game where a player could have an army composed of many unit types: a Partial Player function would then compute the actions of a single unit type.

PartialPlayer function pp $[m = pp(s)]$

- Input state s
- Performs decision logic for a subset of a turn
- Returns partial Move m to perform at state s

The final component of HPS is the portfolio itself which is a collection of Partial Player functions:

Portfolio $P = \langle pp_1, pp_2, ..., pp_n \rangle$

The internal structure of the Portfolio will depend on the game being played. However, it is advised that partial players be grouped by tactical category or game phase. Iterating over all moves produced by partial players in the portfolio can then be performed by the GenerateChildren procedure in Algorithm 6. Once a portfolio is created we can then apply any high-level search algorithm (such as Monte-Carlo tree search or MiniMax) to iterate over all legal move combinations created by the partial players contained within.

5.2.2 State Evaluation

Even with the aid of HPS, games with many turns produce deep game trees which are unfeasible to search completely. We must therefore use a heuristic

Algorithm 6 HPS using NegaMax

```
1: procedure HPS(State s, Portfolio p)
2:     return NegaMax(s, p, maxDepth)
3:
4: procedure GenerateChildren(State s, Portfolio p)
5:     m[] ← ∅
6:     for all move phases f in s do
7:         m[f] ← ∅
8:         for PartialPlayers pp in p[f] do
9:             m[f].add(pp(s))
10:    moves[] ← crossProduct(m[f] : move phase f)
11:    return ApplyMovesToState(moves, s)
12:
13: procedure NegaMax(State s, Portfolio p, Depth d)
14:    if (D == 0) or s.isTerminal() then
15:        Player e ← playout player for state evaluation
16:        return Game(s, e, e).eval()
17:    children[] ← GenerateChildren(s, p)
18:    bestVal ← −∞
19:    for all c in children do
20:        val ← −NegaMax(c, p, d − 1)
21:        bestVal ← max(bestVal, val)
22:    return bestVal
```

evaluation of a game state for use in leaf nodes of the heuristic search. It was shown in chapter 4 that for complex strategy games, formula-based evaluation functions can be used to some success, but are outperformed by evaluations using symmetric game playouts. The concept is that even if the policy used in a playout is not optimal, if both players follow this policy to the end of the game from a given state the winner probably had an advantage in the original state. The Game function is used to perform this playout for evaluation. Finally, an example of HPS using NegaMax as the top-level search algorithm and Game playouts as the heuristic evaluation method can be seen in Algorithm 6.

5.3 Prismata

Prismata is a strategy game developed by Lunarch Studios which combines "concepts from real-time strategy games, collectible card games, and table-top strategy games" [49]. Prismata has the following game properties:

- **Two player**: While Prismata does have single player puzzle and campaign modes, this paper will focus on the more popular and competitive 1 vs. 1 form of Prismata
- **Alternating Move**: Players take turns performing moves like in Chess. However, turns may consists of multiple actions taken by the same player (such as buying units or attacking). The turn is over when the active player declares no additional actions and passes, or a time limit is reached
- **Zero Sum**: The outcome of a game of Prismata is a win, loss or a draw (stalemate), with a winner being declared if they have destroyed all enemy units.
- **Perfect Information**: All players in Prismata have access to all of the game's information. There are no *decks*, *hands*, or *fog of war* to keep information secret from your opponent like in some other strategy games.
- **Deterministic**: At the beginning of a game, a random set units (depending on game type) is added to the base pool of purchasable units. After this randomization of the initial state, all game rules are deterministic.

5.3.1 Game Description

In Prismata, each player controls a number of units and has a set of resources which are generated by the units they control. These resources can then be consumed to purchase additional units which can eventually create enough attack power to destroy enemy units. The main elements and rules of the game are as follows:

- **Units**: Each player in Prismata controls a number of units, similar to a real-time strategy game. Players build up an army by purchasing

Figure 5.1: A screenshot from a typical game of Prismata. The units available for purchase are listed on the left, while the unit instances in play are displayed in the center / right. Units which can block have a blue background, and those that can produce attack have a sword icon in the bottom-left corner.

additional units throughout the game in order to attack the enemy player and defend from incoming attacks. There are dozens of unique unit types in the game, with each player being able to purchase multiple *instances* of each unit type, similar to how a player in a real-time strategy game can have multiple instances of unit such as a tank or a marine. Each unit type in Prismata has a number of properties such as initial hit points, life span, whether or not it can block, etc.

- **Abilities**: Each unit type has a unique set of abilities which allow it to perform specific actions such as: produce resources, increase attack, defend, or kill / create other units. The most basic and important unit of any Prismata game is the Drone, whose ability can be used by the player to produce one gold resource. Unit abilities can only be activated once per turn during the action phase.

- **Resources**: There are 6 resource types in Prismata: gold, energy, red, blue, green, and attack. The gold and green resource types accumulate from turn to turn, while energy, red, and blue are depleted at the end of

a turn. Attack is a special resource and is explained in the next section. Players may choose to consume resources in order to purchase additional units or activate unit abilities.

- **Combat**: The goal of Prismata is to destroy all enemy units. Combat in Prismata consists of two main steps: Attacking and Blocking. Unlike most strategy games, units do not specifically attack other units, instead a unit generates an amount of attack which is summed with all other attacking units into a single attack amount. Any amount of Attack generated by units during a player's turn **must** be assigned by the enemy to their defensive units (blocked) during the Defense phase of their next turn. When a defensive player chooses a blocker with h health to defend against a incoming attack: if $a \geq h$ the blocking unit is destroyed and the process repeats with $a - h$ remaining attack. If $a = 0$ or $a < h$ the blocking unit lives and the defense phase is complete. If a player generates more attack than their opponent can block, then all enemy blockers are destroyed and the attacking player enters the *Breach* phase where remaining damage is assigned to any of the enemy units.

5.3.2 AI Challenges

Prismata is a challenging game to write an AI for, mainly due to its large state and action spaces which create unique challenges for even state-of-the-art search algorithms.

State Space

The state space of a game (how many board positions are possible) is often used as an intuitive measure of game complexity. In Prismata, we can calculate a rough estimate of the state space as follows. In a typical Base + 8 game players have access to 11 base units and 8 random units, for a total of 19 units per player, or 38 in total. If we give a conservative average supply limit of 10 per unit per player, then the number of possible combinations of units on the board at one time in Prismata is approximately 10^{40}. We then have to

consider that each unit can have different properties: can be used or unused, have different amounts of hit points, stamina, or chill, etc. If we give an estimate of an average of 40 units on the board at a time, each with 4 possible states, then we get 4^{40} combinations of properties of those units, or about 10^{24}. Now factor in the fact that Prismata has about 100 units (so far) of which 8 are selected randomly for purchase at the start of the game, and we have about 10^{10} possible starting states in Prismata. In total, this gives a conservative lower bound of 10^{74} as the state space for Prismata.

Action Space

The action space of a game can be a measure of its decision complexity: how many moves are possible from a given state? A turn in Prismata consists of 4 main strategic decisions: defense, activating abilities, unit purchasing, and breaching enemy units. Even if we consider these problems as independent, each of them has an exponential number of possible sequences of actions. Consider just the buying of units: given just 8 gold and 2 energy there are 18 possible ways to buy units from the base set alone. With a typical mid-game resource count of 20 gold, 2 energy, 2 green, 2 blue, and 4 red there are over 25,000 possible base-set combinations of purchases within a turn. Combining all game phases, it is possible to have millions of legal action combinations for a given turn.

Sub-Game Complexity

While state and action spaces are typically used as intuitive indicators of a game's complexity, they do not *prove* that finding optimal moves in a game is computationally difficult. In order to further demonstrate the complexity of Prismata, we show that well known computationally hard problems can be polynomial-time reduced to several strategic sub-components of the game. When deciding which strategic units to purchase, expert players will also attempt to maximize the amount of resources spent on a given turn in order to minimize waste. Given a set of resources and a set of purchasable units with unique costs, the optimization problem of deciding which sequence of

unit purchases sum to the most total spent resources is equivalent to the well known Knapsack problem, which is NP-hard. Also, when deciding how to defend against an incoming attack, expert players will often attempt to let less expensive units die while saving more costly and strategically valuable units. The process of blocking in Prismata involves splitting a total incoming integer attack amount among defenders each with an integer amount of hit points. The optimization problem of determining which blocking assignment leads to the least expensive total unit deaths is a bin-packing problem, which is also NP-hard.

5.4 Prismata AI System

This section describes the Prismata AI system architecture as well as how HPS is applied to Prismata.

5.4.1 AI Environment and Implementation

Prismata is currently written in ActionScript and played in a browser using Flash, which is a notoriously slow language for CPU intensive algorithms. The heuristic search algorithms proposed require the ability to do fast forward simulation and back-tracking of game states. To accomplish this, the Prismata AI system and the entire Prismata game engine were re-written in C++ and optimized for speed. This C++ code was then compiled to a JavaScript library using emscripten [31], resulting in code which runs approximately 5 times slower than native C++, or about 20 times faster than ActionScript. This AI system stays idle in a JavaScript worker thread until it is called by the Prismata ActionScript engine. At the beginning of each AI turn, the ActionScript game engine sends the current game state and AI parameters to the JavaScript AI system, which after the allotted time limit returns the chosen move. This threaded approach allows the AI to think over multiple game animation frames without interrupting the player's interaction with the user interface.

5.4.2 Hierarchical Porfolio Search in Prismata

We will now describe how Hierarchical Portfolio Search is applied to Prismata, which fortunately has some properties which make this method especially powerful. Prismata has 3 distinct game phases: Defense, Action, and Breach, each with their own rules and set of goals. In the defense phase you are trying to most efficiently keep your units alive from enemy attack, in the action phase you are trying to perform actions to generate attack and kill your opponent, and in the breach phase you are trying to most effectively destroy your opponent's units. We can break these 3 phases down even further by considering the action phase as two separate sub-phases: using abilities, and buying units, leaving us with 4 phases. While these phases are technically all part of the same turn, even the best human players often consider them as independent problems that they try to solve separately, as the entire turn would be too much to mentally process at the same time. We then develop a number of algorithms (Partial Players) for attempting to choose good actions for each individual phase. For example, in the defense phase we could have one Partial Player that tries to minimize the amount of resources you will lose if you block a certain way, while another would try to maximize the amount of attack you have remaining to punish your opponent with.

Portfolio $P = \langle PP_1, PP_2, PP_3, PP_4 \rangle$

> A set of Partial Players PP_i corresponding to each of the four phases described above

This portfolio of Partial Players for each phase will now serve as a move iterator for our high-level search algorithm to search over all combinations of each move for each phase in order to determine the best move for the turn. Once the portfolio move iterator has been constructed, we use a high-level search algorithm to decide which move combination to perform. The search algorithms used for the Prismata AI system are UCT [46] and Alpha-Beta with iterative deepening.

5.4.3 AI Configuration and Difficulty Settings

All AI components in Prismata can be modularly described at a very high level in a text configuration file. This enables easy modification of all AI components quickly and intuitively without the need to modify code or even recompile the system. All components of the system can be modified in the configuration: Players, Partial Players, Portfolios, Search Algorithms, States, and Tournaments. These components are arranged in a dictionary with a description of the component as the key and collection of parameters as its value. Partial Players are arranged via tactical category and can be combined in any order to form full Players or Portfolios. Search algorithm parameters such as search depth, time limits, evaluation methods, and portfolio move iterators are also specified here. Player specifications can also quickly be arranged to play automatic AI vs. AI tournaments for strategic evaluation, code benchmarking, or quality assurance testing.

Using the search configuration syntax, creating different difficulty settings for the Prismata AI is trivial. After the hardest difficulty had been created (Master Bot - using Monte-Carlo Tree Search), five other difficulty settings were then created: Docile Bot (never attacks), Random Bot (random moves), Easy Bot (makes moor defensive choices), Medium Bot (makes poor unit purchase choices), and Expert Bot (performs a 2-ply alpha-beta search). All of these difficulties were created in less than 15 minutes simply by creating new combinations of Partial Players within the AI settings file. While only the Expert and Master difficulty settings use the high level search system of HPS, the others were still created within the overall HPS architecture.

5.5 Experiments

Several experiments were performed to evaluate the proposed AI architecture and algorithms. All computational experiments were performed on an Intel i7-3770k CPU @ 3.50GHz running Windows 7.

5.5.1 AI vs. Human Players

Prismata's most competitive format is its ranked ladder system in which human players get paired against similar skilled opponents through a automated match-making system. Player skill is determined via a ranking system in which players start at Tier 1 and progress by winning to Tier 10, at which point players are ranked within tier 10 with an ELO-like numerical rating. To test the strength of the AI vs. human opponents in an unbiased fashion, an experiment was conducted in which the AI was configured to secretly play games in the human ranked matchmaking system over the course of a 48 hour period. Going by the name "MyNameIsJeff", the AI system was given randomized clicking timers in order to more closely resemble the clicking patterns of a human player. The AI player used was the in-game Master Bot, which used UCT as its high-level search with a 3 second time limit. During the period the AI played approximate 200 games against human opponents with no player realizing (or at least verbalizing) that they were playing against a computer controlled opponent. After the games were finished, the bot achieved a ranking of Tier 6 with 48% progression toward Tier 7. The distribution of player tier rankings at that time is shown in Table 5.1, placing the bot's skill within the top 25% of human players on the Prismata ranked ladder. It is estimated by expert players of Prismata that the updates to the AI system since this experiment was done now place it around rank 8.

5.5.2 Difficulty Settings

Two experiments were performed to test the playing strength of various difficulty settings of the Prismata AI bots. The first experiment was conducted to test if the playing strength rank of the various difficulty settings matched their descriptive rank. Descriptions of each bot difficulty are as follows:

- **Master**: Uses a Portfolio of 12 Partial Players and does a 3000ms UCT search within HPS, chosen as a balance between search strength and player wait time

- **UCT X**: Uses the same Portfolio as Master bot, does an X millisecond UCT search within HPS
- **AB X**: Uses the same Portfolio as Master bot, does an X millisecond Alpha-Beta search within HPS
- **Expert**: Uses the same Portfolio as Master Bot, does a 2-ply fixed depth alpha beta search within HPS
- **Medium**: Picks a random move from Master Bot's Portfolio
- **Easy**: Medium, but with weaker defensive purchasing
- **Random**: All actions taken are randomly chosen until no more legal actions remain and the turn is passed

Both UCT and Alpha-Beta were chosen as the high-level search algorithms for HPS, and in order to demonstrate the performance of HPS under short time constraints their time limits were set to 100ms per decision episode. 10,000 games of base set + 8 random units were played between each pairing, with a resulting score given for each pairing equal to win% + (draw%/2). The results for this experiment are shown in Table 5.2 and show that the difficulties do indeed rank in the order that they were intended. It also shows that at short time controls both UCT and Alpha-Beta perform equally well.

The second experiment tested the relative performance of UCT and Alpha-Beta at different time settings in order to determine how an increase in thinking time affects playing strength. 1,000 games of base set + 8 random units were played between all pairings of Alpha-Beta and UCT, each with time limits of 3000ms, 1000ms and 100ms. Results are shown in Table 5.3 and indicate that playing strength increases dramatically as more time is given to each search method. An interesting note is that Alpha-Beta outperforms UCT at longer time limits. We believe that this is in part caused by the fact that all players use the same portfolio as the basis for their move iteration, therefore Alpha-Beta may have an advantage over our UCT implementation which does not yet perform sub-tree solving.

Table 5.1: Prismata Player Ranking Distribution

Tier	1	2	3	4	5	6	7	8	9	10
Player Perc.	33.9	17.3	7.1	7.5	6.7	7.5	6.5	5.9	3.7	4.0

Table 5.2: Search vs. Difficulties Results (Row Win %)

	UCT100	AB100	Expert	Medium	Easy	Rnd.	Avg.
UCT100	-	52.1	67.3	96.4	99.7	99.9	83.1
AB100	47.9	-	68.0	94.7	99.5	99.9	82.0
Expert	32.7	32.0	-	90.7	98.9	99.8	70.8
Medium	3.6	5.3	9.3	-	85.9	97.4	40.3
Easy	0.3	0.5	1.1	14.1	-	86.3	20.5
Random	0.1	0.1	0.2	2.6	13.7	-	3.3

Table 5.3: Search Algorithm Timing Results (Row Win %)

	AB3k	UCT3k	AB1k	UCT1k	UCT100	AB100	Avg.
AB3k	-	58.9	64.5	66.8	83.8	85.2	71.8
UCT3k	41.6	-	53.9	65.3	81.1	81.5	64.7
AB1k	35.5	46.3	-	58.1	76.3	80.2	59.3
UCT1k	33.4	34.8	41.9	-	70.1	74.1	50.9
UCT100	16.0	18.7	23.6	29.7	-	53.4	28.3
AB100	14.5	18.3	19.5	25.6	46.3	-	24.8

5.5.3 User Survey

A user survey was conducted to evaluate whether or not the design goals of the Prismata AI system had been met from a user perspective. The following questions were asked about the user's experience with the Prismata AI bots, with each answer was numerical on a scale from 1-7:

1. How has your overall experience been so far with the Prismata bots? (1 = Not Enjoyable, 7 = Very Enjoyable)
2. How would you rate the Prismata bots as a tool for new players to learn the basic rules / strategies of the game? (1 = Bad Tool, 7 = Good Tool)
3. How would you rate the Prismata bots as a tool for experienced players to practice strategies / build orders? (1 = Bad Tool, 7 = Good Tool)
4. How does the difficulty of the Prismata AI compare to the AI in similar games you have played? (1 = Much Weaker, 7 = Much Stronger)
5. Do you think the difficulties of the Prismata bots match their described skill level? (1 = Poor Match, 7 = Good Match)
6. How does the overall experience of the Prismata AI compare to the AI in similar games you've played? (1 = Less Enjoyable, 7 = More Enjoyable)

In each question we consider a mean score of greater than 4 (the median) as a success. After running for 10 days online, the survey received 95 responses, with the results shown in Table 5.2. Overall the survey response was very positive with users ranking their overall experience in the Prismata AI with a mean of 5.55 out of 7 which is quite enjoyable. Users responded that the Prismata AI system's strength was higher than that of similar games they had played with a mean of 5.43, and that their overall experience with the Prismata AI was more enjoyable than their experiences with the AI in similar games with a mean of 5.47. Users felt that the Prismata AI bot difficulty settings matched their described skill level with a score of 4.86, which is overall positive but leaves much room for improvement. Users rated the Prismata AI as a very

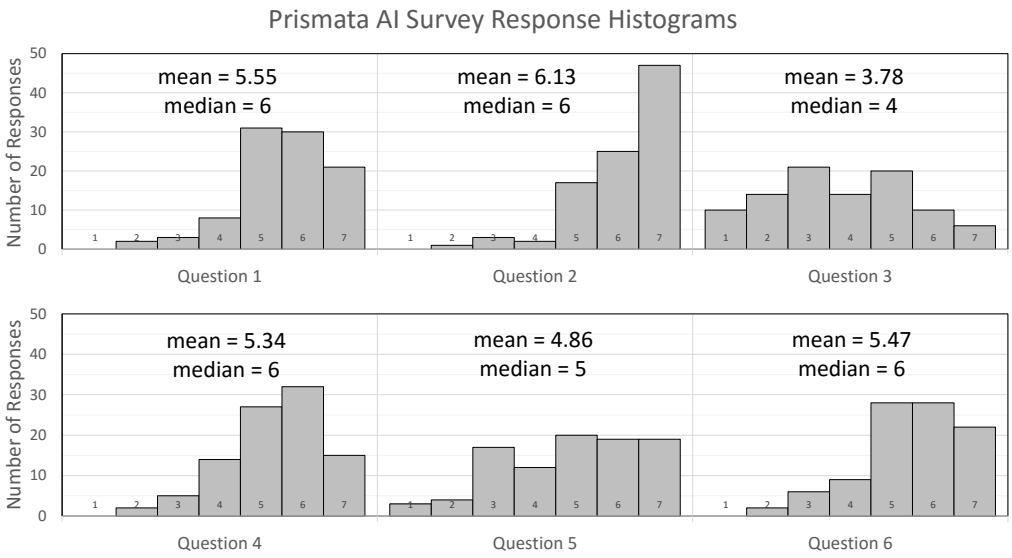


Figure 5.2: Result histograms from the Prismata AI Survey, with 95 responses total. Shown for each question are the number of responses for each value from 1 to 7.

good tool for new players to learn the game with a mean of 6.13, but had mixed responses about its use as a tool for experienced player practice, with a mean of 3.78. While the AI ranked in the top 25% of player skill, expert players are able to beat the AI 100% of the time meaning that it is not yet a good candidate for expert practice. We feel that these survey responses show that from a user perspective, the Prismata AI experience is a success, and was able to meet the specified design goals.

5.6 Summary

In this chapter we presented several design goals for AI systems in modern video games, along with two main contributions to try and meet those goals. The first contribution was Hierarchical Portfolio Search, a new algorithm designed to make strong strategic decisions in games with very large action spaces. The second was the overall AI architecture which incorporated Hierarchical Portfolio Search and was used for the strategy game Prismata by Lunarch Studios. This AI system was played in secret on the ranked human

ladder and achieved a skill ranking in the top 25% of human players, showing that HPS was successful in creating a strong playing agent in a real-world video game. Users were then surveyed about their experiences with the Prismata AI system and responded that they felt the game's AI was stronger and the overall experience was better than in similar games they had played. In the past 14 months that this AI system has been in place no architectural changes or significant AI behaviour modifications were required, despite dozens of individual unit balance changes being implemented by the game's designers, proving its robustness to such changes.

Future work with the Prismata AI system will be focused on improving bot strength in an attempt to reach a level similar to that of expert players. Not only will this provide a more valuable tool for experienced player practice, but it could also be used as a tool for future research in automated game design and testing. If an AI agent can be made that is able to play at the level of expert players, the process of game balance and testing could then be automated instead of relying solely on human players for feedback. For example, if a designer wants to test a new unit design before releasing it to the public they could run millions of AI vs. AI games in an attempt to see if the unit is purchased with the desired frequency or if it leads to an imbalance in win percentage for the first or second player. This will not only reduce the burden on designers to manually analyze new unit properties but also reduce player frustration if an imbalanced unit is released for competitive play. We hope that in the future artificial intelligence will play a much greater role in the game design process, reducing development time and providing useful tools for designers and testers so that more enjoyable experiences can be delivered to players more quickly and easily than ever.

Chapter 6

Software Contributions

In this chapter we will discuss the various software contributions which have been implement as a result of the research presented in this thesis.

6.1 UAlbertaBot

UAlbertaBot is the University of Alberta's STARCRAFT AI competition bot, which I have written and maintained since 2011. UAlbertaBot has regularly placed at the top of every major international STARCRAFT AI competition, and was the winner of the 2013 AIIDE competition. The design and goals of the UAlbertaBot project have evolved over the past five years into an easy to use, robust, modular system capable of playing any of the three STARCRAFT races. UAlbertaBot has been the basis of several other well-performing competition bots such as LetaBot [66] which won the 2014 and 2015 Student STARCRAFT AI Tournament [16] student division. UAlbertaBot is also used as an educational tool in the CMPUT350 Advanced Games and AI Programming course at the University of Alberta. In this section we will describe the moudlar architecture of UAlbertaBot, the AI techniques that it uses, and the overall logic flow of the bot, and give a brief history of some of the milestones it has achieved.

6.1.1 Design

UAlbertaBot is written in C++ using BWAPI with a hierarchical and modular design, similar to a military command structure. A class diagram of UAlbertaBot can be seen in Figure 6.1. This modular design is also quite

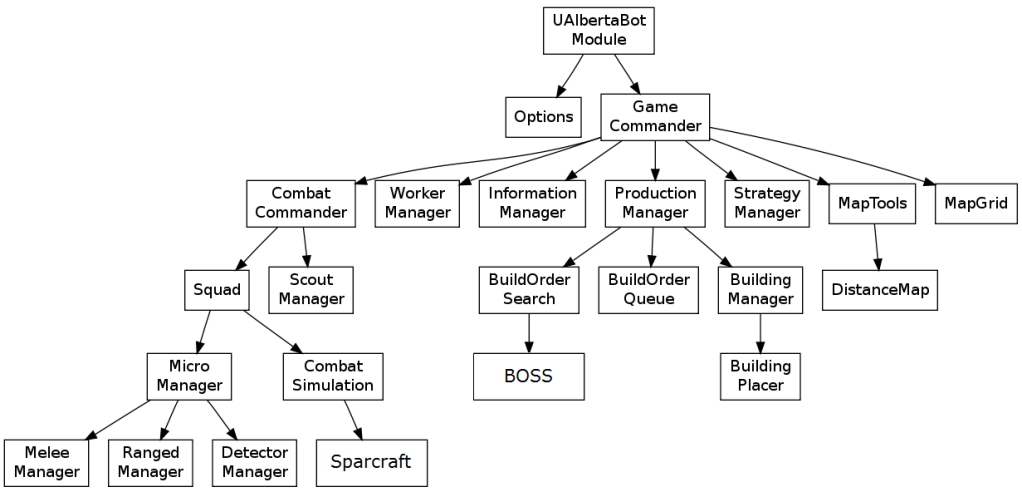


Figure 6.1: Class diagram of UAlbertaBot.

important from a software design and implementation standpoint, as it allows sections of the bot to be easily upgraded as new solutions are developed. For example, the build-order planning module can be thought of as a function which takes as input the current STARCRAFT state, and an army composition, and as output produces a build-order action sequence. Initially, this was implemented as a rule based system, but based on the research described in chapter 3 it was replaced with a search-based build-order planner without affecting any other code in the bot. Similarly, an initially rule-based combat timing system in Combat Commander module was replaced with the SparCraft simulation package described in chapter 4 in order to automatically determine when units should attack or retreat. The modular design has also made it easy for other people to use UAlbertaBot and to modify it use different strategies.

A full class diagram of UAlbertaBot can be seen in figure 6.1. The UAlbertaBot module is the main module of the bot which is used by BWAPI to construct the .dll which is injected into STARCRAFT when the game is launched. When the bot is first started, this module parses the bot configuration file and stores the options globally which are used by all of the other modules. After the initial starting of the bot, BWAPI interacts with the STARCRAFT game engine and calls this module's OnFrame() function after each logical frame of STARCRAFT has finished. The OnFrame() function can be thought of as a standard main loop, with all of the logic for UAlbertaBot happenening inside it.

Once the OnFrame() function has finished, BWAPI passes control back to the STARCRAFT engine which executes all commands given during the OnFrame() function of the bot, and then the process repeats until the game is over. The OnFrame() function of the UAlbertaBot module calls the GameCommander's OnFrame() function, which performs all the strategic logic for the bot. The sequential logic flow for each of the modules can be seen in figure 6.2.

6.1.2 Strategy and AI Systems

UAlbertaBot is designed to be able to play all three races in STARCRAFT: Protoss, Terran, or Zerg, and to be able to implement any strategy for those races with little modification. UAlbertaBot contains several AI systems which are run in real-time during competition settings and are speficically designed to facilitate the playing of any of the races. These AI systems are as follows:

- **Build-Order Planning**: UAlbertaBot uses the build-order planning system described in chapter 3. All build-orders for UAlbertaBot are planned online in real-time, with two small exceptions: a supply producer is build immediately if a supply block is detected, and a detector is immediately built if an enemy cloaked unit is detected. This system was the first real-time search based planning system used in a STARCRAFT competition.

- **Combat Simulation**: UAlbertaBot typically implements a rush, which is a very early attack strategy. Typically when humans implement a rush strategy, they must determine a time or a number of army units to obtain before they begin their attack. UAlbertaBot instead simply sends any army units to attack the opponent base as soon as they are produced, and relies on the battle outcome simulation in SparCraft as a measure of whether or not it should continue toward the enemy base or retreat and regroup. This system was the first full battle combat simulator used in real-time in a STARCRAFT competition.

- **Strategy Definition**: A strategy in UAlbertaBot consists of two major

GameCommander.OnFrame()

- WorkerManager.update()
  - Re-allocate workers to new mineral patches if mined out
  - Assign all currently idle workers to gather minerals
  - Allocate gas workers until 3 workers are at each refinery
  - Move constructing workers to their building locations
- ProductionManager.update()
  - If event occurs which triggers a new build order
    - Clear current build order
    - Get new build order goal from Strategy Manager
    - Start new build order search search in for the current goal
  - Build the highest priority item in the build order queue if possible
  - If item to be built is a building, add task to Building Manager
- BuildingManager.update()
  - Check if any workers assigned to build have died before finishing
  - Find building placement by spiraling outward from desired location
  - Assign workers to unassigned buildings and label them as planned
  - For each planned building, tell the worker assigned to construct
  - Continuously monitor the status of buildings under construction
  - If we are Terran and worker died mid construction, assign another
  - If a building completes, remove the task and mark worker idle
- CombatCommander.update()
  - Set a a worker scout if it is the appropriate time to scout
  - If any enemies are near one of our bases, assign a defense squad to it
  - If any additional units are available to attack, attack as follows:
    - If an enemy base location is visible, attack it
    - Otherwise, if we see visible enemy units, attack them
    - Otherwise, if we know the location of an enemy building, attack it
    - Otherwise, explore the least recently seen region of the map
  - Squads.update() - perform all squad logic and commands
    - If defense force killed enemy attackers, add defenders to attack squad
    - Perform combat simulation with SparCraft for attack squad
    - If simulation predicts victory, continue attacking
    - If attack continues, call MicroManager sub-class for individual unit control
    - If simulation predicts defeat, retreat units toward home base
    - If squad contains no units, delete it
- ScoutManager.update()
  - If we know where the enemy base is, go toward it and continue observing it
  - If scout is not being attacked, attack the closest enemy worker, retreating if damaged
  - If we can't see enemy base, explore the closest known possible base location
- InformationManager.update()
  - If an enemy unit is visible, record the last known location it was seen
  - If an enemy unit dies, record the resource loss to predict their current resource total

Figure 6.2: Sequential logic flow for UAlbertaBot.

components: the opening build order (if one is desired) and the build-order goal decision making. A new strategy can be added to the bot by modifying the opening build order in the configuration file, and modifying the build-order goal decision function in the bot's StrategyManager module. This function is currently hard-coded, and reads the current game state to determine what units the bot should produce next. For example, if the bot is performing a Zealot rush, the returned goal will consist of some number of Zealots higher than currently owned. This goal is then searched for by the build order search system. Other strategic units are also instered into the build order goal, such as adding a detector unit to the goal when invisible enemy units are scouted. UAlbertaBot currently has 10 stratgies in total including aggressive strategies such as Zergling, Zealot, and Marine rushes, sneaky strategies such as Dark Templars or Zealot Drops, and some late game strategies such as Zerg Mutalisks, Zerg Hydralisks or Terran Wraiths. New strategies can be added to the bot in a matter of minutes.

- **Strategy Selection**: Since the bot can perform many different stratgies, we must decide on which strategy to use somehow. In the 2012 AIIDE competition, UAlbertaBot used persistent storage to record the results of previous match outcomes, and then used UCB at the beginning of future matches to determine which strategy to use. For the 2015 AIIDE competition, UAlbertaBot played hundreds of games against existing bots and determined which stratgies worked well against them, implementing a number of opponent modeled strategies against specific bots by name. For example, if UAlbertaBot received the Terran race and its opponent was Ximp it would implement a Tank push strategy since it knew the previous version of Ximp was quite weak to it. UAlbertaBot can decide on strategies in any of these ways: Learning via UCB-1 or Epsilon Greedy based on previous match results in a tournament, it can define specific strategies to use if the enemy has a specific name, strategies can be selected randomly, or hard-coded to be used for each game.

- **Multi-Agent Pathfinding**: UAlbertaBot uses a 4-directional flood-fill algorithm to compute all approximate shortest paths to a single goal. These paths are then cached in memory so that future paths to the same goal do not need to be re-computed.

- **Other Systems**: All other AI components in the bot are currently implemented as rule-based scripts.

6.1.3 Competition Results and Milestones

UAlbertaBot has competed in every major STARCRAFT AI Competition since the first AIIDE competition in 2010. A complete listing of UAlbertaBot's results can be found in table 6.1. UAlbertaBot's major research and competition milestones are as follows:

- **2010**: First version of UAlbertaBot is created by David Churchill and Sterling Oersten is created for the 2010 AIIDE STARCRAFT AI Competition. UAlbertaBot initially played the Zerg race and implemented a Mutalisk strategy. The competition was single elimination random pairings and the bot lost to krasi0 in the third round, which ended up coming 2nd overall in the competition. The bot consisted mainly of one hard-coded strategy implemented with the BWSAL library.

- **2011**: UAlbertaBot was completely re-written from the ground up due to poor architectural decisions made in the first implementation. The bot was changed to play the Protoss race and implement an aggressive Zealot Rush strategy. The BOSS system described in chapter 3 was integrated into the bot which was able to dynamically plan all build-orders in real-time, which was the first such system ever used in a competition setting. This new UAlbertaBot placed 2nd in both the AIIDE and CIG competitions, which were both won by Skynet - another Protoss bot whose solid early game defense was able to hold off the aggression of UAlbertaBot.

- **2012**: UAlbertaBot implemented two new strategies on top of the existing Zealot rush: Dragoon rush, and Dark Templar (invisible unit) rush. Persistent file IO became available in the 2012 AIIDE competition, so UAlbertaBot recorded the results of each game and used the UCB-1 algorithm to select which strategy to use at the start of the next game. The SparCraft combat simulation system described in chapter 4 was integreated into UAlbertaBot which provided the ability to predict the outcome of combat skirmishes, which greatly increased the bot's combat efficiency. UAlbertaBot placed 3rd at AIIDE, 2nd at CIG, and 3rd at SSCAIT with these new updates. Skynet again won the AIIDE and CIG competitions.

- **2013**: After inspecting the 2012 competition results, it was evident that the Dragoon and Dark Templar strategies were not as strong as the Zealot rush strategy, and the games that UCB-1 spent exploring those strategies were essentially all losses. In 2013 UAlbertaBot reverted back to a single Zealot rush strategy with improved timing and an updated version of the SparCraft combat simulator. Several small bug fixes and early-game strategy adjustments were also implemented, which resulted in UAlbertaBot winning the 2013 AIIDE competition.

- **2014**: UAlbertaBot was not upgraded in 2014 and so the 2013 version was submitted to each competition. Since it had won the 2013 competition, many bots implemented hard-coded strategies against it and so it performed relatively poorly in the 2014 competitions and 2015 CIG competition.

- **2015**: UAlbertaBot underwent major architectural and strategic changes in 2015 which were completed after the 2015 CIG competition but in time for the 2015 AIIDE competition. The biggest change was implementing a more generalized AI architecture so that the bot could now play any of the 3 races instead of just playing Protoss. UAlbertaBot played the Random race for the AIIDE 2015 competition, which was the first time

Competition	Rank	Entrants	Games	Wins	Losses	Race
2015 AIIDE	4	22	1889	1515	374	Random
2015 CIG	10	14	390	189	201	Protoss
2015 SSCAIT	3	46	45	34	11	Random
2014 AIIDE	7	18	1139	766	373	Protoss
2014 CIG	5	13	720	432	288	Protoss
2014 SSCAIT	3	42	41	32	9	Protoss
2013 AIIDE	1	8	1393	1177	216	Protoss
2013 CIG	2	8	?	?	?	Protoss
2013 SSCAIT	3	50	?	?	?	Protoss
2012 AIIDE	3	10	1656	1136	520	Protoss
2012 CIG	2	10	?	?	?	Protoss
2012 SSCAIT	3	52	?	?	?	Protoss
2011 AIIDE	2	13	360	286	74	Protoss
2011 CIG	2	10	70	55	15	Protoss
2010 AIIDE	>4	17	?	?	?	Zerg

Table 6.1: UAlbertaBot results for major STARCRAFT AI Competitions. Question mark indicates values that are unknown or not applicable.

a bot had played Random race in any major competition. UAlbertaBot was upgraded to have a total of 10 different strategies, with several for each race including: Protoss Zealot rush, Protoss Dark Templars, Protoss Dragoons, Terran Marines, Terran Bunker-First, Terran Vultures, Terran Tank Push, Zergling Rush, Zerg 3-Hatch Hydralisk, and Zerg Anti-Air. UAlbertaBot was trained against the 2014 versions of many top performing bots prior to AIIDE 2015, which allowed it to select strategies against individual bots. UAlbertaBot ended up placing 4th overall at AIIDE 2015, however it should be noted that it actually had a winning record against each other bot in the competition. Its inability to exploit some of the weaker bots as much as the top 3 bots resulted in slightly lowering its overall win percentage and placing 4th.

6.1.4 Impact and Research Use

UAlbertaBot was designed from the ground up to be not only a top performing competition bot, but a modular and easy to use tool for RTS AI research. UAlbertaBot is actively maintained as an open source project hosted on GitHub

[22] with full documentation, installation guide, and video coding tutorials, making it very easy to download and use by programmers of any skill level. The SparCraft combat algorithms and simulation system described in chapter 4, as well as the Build-Order Search System described in chapter 3 are also available as open source projects on GitHub [21] as part of the UAlbertaBot project. Over the years, hundreds of researchers, students, and hobbysists have used these projects for implementing experiments, course projects, or as the basis for their competition bots.

All or part of the UAlbertaBot code base has been used as the basis for several STARCRAFT AI competition bots, including: LetaBot [66], Overkill [91], TerranUAB [7], NUSBot [93], MooseBot [54], Odin [51], Bonjwa, HITA, and Chris Ayers unnamed 2015 SSCAIT entry. UAlbertaBot and SparCraft have been used for the experimental results in recent publications on topics such as predicting RTS combat outcomes [76] [74], learning RTS combat models [85], global RTS game state evaluation [32], build placement optimization [4], high level strategy search [5], hierarchical adversarial search [75], cluster-based RTS combat [44], and unit motion analysis [71]. UAlberaBot has also been used by students as an educational tool. Since 2012, the CMPUT 350 course at the University of Alberta has used UAlbertaBot for its final course project, where undergrads modify UAlbertaBot's AI systems to create new strategies and tactics.

6.2 Tournament Manager Software

As STARCRAFT AI tournaments grew in popularity, it became obvious quite early on that running bot vs. bot matches by hand was a tedious and cumbersome process, often involving several minutes of set up time per game and only feasible for playing a single match at a time. In 2011, the University of Alberta took over organizing the AIIDE STARCRAFT AI competition, which has been organized and run by myself and Michael Buro every year since. Jason Lorenz and I wrote software to automate the process of running STARCRAFT AI tournaments, and I have been actively maintaining it by myself

since 2012. As of winter 2014, all three major Starcraft AI competitions (AIIDE, CIG, and SSCAIT) use this software to play their tournaments, with over 50,000 competition games having been played using the software. Not just a tool for running competitions, this software allows users to play a single bot against many other bots and collect the detailed results automatically, which is very useful for bot developers when trying to analyze their bot's performance against various opponents, collect statistics for research papers, or to help automatically debug performance issues. In this section we will briefly discuss the design, architecture, and implementation of the software. The software is written in Java, and is split into two main components: the server, and the client.

6.2.1 Server

When running the software, one machine acts as a server for the tournament. The server is a central repository where all bot files (including file I/O) data, cumulative results, and replay files are stored. The server also monitors each client remotely and outputs status and results to an html file so that tournament status can be viewed in real time. The server program has a threaded component which monitors for new client connections and detects client disconnections, maintaining a current list of clients which can have one of the following statuses: READY - Client is free and ready to start a game of STARCRAFT, STARTING - Client has started the STARCRAFT LAN lobby but the match has not yet begun, RUNNING - Client is currently running a game of StarCraft, SENDING - Client has finished the game and is sending results and data back to the server. When a client initially connects to the server, the server sends it the Chaoslauncher program automatically.

When the server is started, it first reads the server settings file which contains information such as the port to run on, and the names and details of all the bots in the competition. It checks to see if all the required bot directories exist, whether or not their persistent storage folders exist, and whether or not the required dll files are present in those directories. When those conditions are met, it then checks if any current game schedule exists. I

no game schedule (called the game list) exists, it will ask the user to create a new one by specifying the number of rounds of round robin that they wish to play between all the bots. Users can manually create a game list file to play any type of tournament, with the syntax simply being the names of the bots to be played and the map to play on, which are played in the order listed in the file. The bot then parses the current results file if it exists, and skips any games which already have results recorded, which allows the tournament to be started and stopped at any point without losing the results of any games which were previously played.

The server's main scheduling loop then activates, attempting to schedule the next unplayed game from the games list every 2 seconds. A new game can be started only if two or more Clients are READY, and no clients are STARTING. The reason no clients can be STARTING is to prevent multiple STARCRAFT game lobbies from being open on the same LAN, which may cause mis-scheduled games due to limitations on how STARCRAFT/ BWAPI are able to join games on a given network. Once these two conditions are met, the server sends the required bot files, map files, and chaoslauncher configuration to the client machines, specifying one client as the host and one as the away machine. All files are compressed and sent via Java sockets, which ensures that the software is compatible with any network that supports them. Once all files have been received by the client, those clients' status are then set to STARTING. Each client is handled by a separate thread in the server, and if the client is STARTING, RUNNING, or SENDING, it sends periodic status updates back to the server for remote monitoring. Data such as current game time, time-out information, map, game ID, etc are each updated once per second from each client to the server GUI. When a client finishes a game the results are sent back to the server along with file I/O data and replay files, which are all stored on the server. This process repeats until the tournament has finished. Shutting down the server via the GUI will send a message to all clients to stop all running games, shut down, and clean up properly. The tournament can be resumed upon re-launching the server program as long as the results file and games list do not change.

The server supports persistent file storage so that the bots can write data to later be read for purposes such as strategy learning, with the files being stored relative to each bot on the server machine. After each game finishes on a client machine, the contents of that client's 'write' folder (files output by the bot during the match) are copied to that bot's 'write' folder on the server machine. Whenever a game is scheduled to be run on a client machine, the contents of the bot's 'read' folder from the server are sent to the client machine and extracted there for the bot to read. For the first round of the competition this 'read' folder is initially empty. After each round of round robin is finished, the contents of the bot's 'write' folder on the server is copied into the 'read' folder on the server. This means that the bot will have access to all data written from previous rounds of the competition. By copying data after each round has completed we ensure that no bot has an information advantage by having had more games scheduled than its opponent during a given round. After each game is played, the tournament results are automatically updated and output to HTML files including real-time results tables and charts as the competition progresses.

6.2.2 Client

The client software can be run on as many machines (physical or virtual) that are avaialable on a given local area network, with the only requirement that it supports both TCP (for the Java socket connection) and UDP (which STARCRAFT uses for network play), and that only one client is run per machine. After an initial setup of the client machine (installing STARCRAFT, etc), running the client software connects to the server machine to await instructions. Upon initially connecting to the server, the client recieves the Chaoslauncher program and automatically updates the Windows registry with the required STARCRAFT and Chaoslauncher settings.

The client machine will stay idle until it receives instructions from the server that a game is to be run. Once the client receives the required files from the server, it ensures that no current STARCRAFT processes are running, records a current snapshot of the running processes on the client ma-

chine, writes the BWAPI settings file for Chaoslauncher, and starts the game. When the game starts, a custom BWAPI Tournament Module is injected via Chaoslauncher which outputs a GameState file to disk every few frames, which monitors the current state of STARCRAFT. The client software reads this file to check for various conditions such as bot time-outs, crashes, no game frame progression, and game termination. As the game is running, the client sends the contents of the GameState file to the server once per second to be monitored on the server GUI. Once the game has terminated for any reason, the results of the game, replay files, and file I/O data are sent back to the server. Once the sending is complete, the client software shuts down any processes on the machine which were not running when the game began, to prevent things like crashed proxy bots or stray threads from hogging system resources from future games. STARCRAFT is shut down, the machine is cleaned of any files written during the previous game, and the client status is reported back to the server as READY. The client is then ready to be given a new game to play by the server.

Chapter 7

Conclusion

In this chapter we will give an overview of the contributions made in this thesis, followed by a discussion of promising future topics for research.

7.1 Contributions

7.1.1 Build-Order Optimization

In chapter 3 we presented a depth-first branch and bound algorithm for tackling the problem of build order optimization in real-time strategy games. When combined with several heuristics, this algorithm was capable of finding build orders in real-time for sets of goal units extracted from professional human replays. The resulting build orders from this system produced plans whose makespans were on average about 10% shorter than those of professional human players which computing them in real-time. This system was integrated into UAlbertaBot, our STARCRAFT AI Competition entry, and was the first system in the world to dynamically plan build orders in real-time during a competition setting. This Build Order Search System (BOSS) was released as an open source project several years ago, and has been used by several competition bots as described in section 6.1.4. In the five years since its release, BOSS has still not been outperformed by any other system in terms of generated makespan length.

7.1.2 RTS Combat Micromanagement

In chapter 4 we presented an RTS combat simulator named SparCraft, along with a number of different algorithms for deciding actions for combat scenarios in real-time strategy games. These algorithms were Alpha-Beta Considering Durations (ABCD), UCT Considering Durations (UCT-CD) and completely new Portfolio Greedy Search. In the experiments performed, each of these algorithms were shown to defeat existing state of the art scripted solutions in nearly 100% of the time. When battles sizes were small (8 vs 8 units and under) ABCD outperformed the other two algorithms, while UCT-CD outperformed the other two algorithms in medium sized battles of 8-16 units. For battles larger than 16 vs 16 units, Portfolio Greedy Search was a clear winner, vastly outperforming the state of the art for large-scale RTS combat scenarios.

SparCraft was integrated into UAlbertaBot as a combat simulation tool and is used for battle outcome prediction, which has proven instrumental in UAlbertaBot's success over the past few years of competition - with results dramatically improving after its inclusion in the bot. SparCraft has also been released as an open source project and has been used by several researchers and bot programmers in their own projects, as described in section 6.1.4. The publication introducing SparCraft [21] won the Best Paper award at the 2013 Computational Intelligence in Games (CIG) conference and was invited to be presented at Game Developer's Conference (GDC) 2014.

7.1.3 Hierarchical Portfolio Search

In chapter 5 we introduced Hierarchical Portfolio Search (HPS), a new algorithm for decision making in games with extremely large search and action spaces. HPS was used as the basis for the AI system in Prismata, a hybrid strategy retail video game by Lunarch Studios. HPS greatly reduces the action space of a game by only considering moves generated by a portfolio of sub-algorithms, rather than all possible action combinations. HPS also allows for the easy creation of various difficulty settings simply by modifying the internal portfolio, leading to increased replayability and lower development times for

designers. Experiments showed that HPS produced an AI system for Prismata that reached a skill level within the top 20% of human players on the Prismata ranked ladder. A user survey was also conducted in which players stated that the Prismata AI was more intelligent than similar games they had played, was a very good tool for new players to learn the game, and was overall a better experience than they had had with AI systems in similar games. The publication which introduced HPS [25] won the Best Student Paper award at AIIDE 2015 and was invited to be presented at GDC 2016.

7.1.4 Software Contributions

In chapter 6 we discussed several open source software projects related to RTS game AI, the most important of which being UAlbertaBot [22], our STARCRAFT AI competition entry. UAlbertaBot has consistently placed among the top few bots in all major STARCRAFT AI competitions since 2011, and won the 2013 AIIDE STARCRAFT AI competition. It was the first bot to implement a real-time build-order search system (BOSS) as well as a real-time combat simulation system (SparCraft). By using a robust and modular AI architecture, UAlbertaBot was the first bot to enter a major AI competition as the Random race, achieving a winning record against all other bots in the 2015 AIIDE STARCRAFT AI competition.

UAlbertaBot has been actively maintained as an open source project for the past several years and has been downloaded hundreds of times. It has been used as the basis for many top performing STARCRAFT AI competition bots such as LetaBot [66], which won the 2014 and 2015 Student STARCRAFT AI Tournament [16] student division, Overkill, [91] which placed 3rd in the 2015 AIIDE STARCRAFT AI Competition, and many others as mentioned in section 6.1.4. UAlbertaBot has also been used as an educational tool by the CMPUT 350 course at the University of Alberta, in which undergraduate students modified UAlbertaBot in various ways for their final course project.

The state-of-the-art algorithms and open-source software projects we have created based on the research presented in this thesis, namely: UAlbertaBot, SparCraft, BOSS, and the STARCRAFT AI tournament manager software,

have proven invaluable for many researchers, students, and hobbyists alike. Whether it be writing their own STARCRAFT AI bot, carrying out experiments for research, or running a STARCRAFT AI competition, these tools have aided many people in their work, and helped significantly lower the barrier of entry to the complex field of RTS AI.

7.2 Directions for Future Research

7.2.1 "Goal-less" Build-Order Search

In chapter 3 we introduced an algorithm for performing real-time build-order search in RTS games. While this algorithm performed quite well, it still relied on an outside source providing it the goal set of units that it was trying to achieve. Our idea for future work is to perform a build-order search which does not attempt to achieve a given goal set of units, but instead attempts to maximize a given *army value*. This army value could anything from a simple sum of resources spent on combat units, to the result of a complex combat search algorithm as presented in chapter 4. This new algorithm could adapt more easily to a given game setting without the expert knowledge which is currently required to construct the set of goal units.

We have obtained some preliminary results in this direction which hold some interesting insight into this problem. For these results, we have taken the simplest possible formula for an army value which we attempt to maximize, which is the total sum of resources spent on combat units in our produced army. The first intuition we had for performing this new goal-less search was to find a build-order for which the army value was maximized for some future time in the game. In Fig. 7.1, the green line shows the army value at each time step for the single build-order which maximizes the army value at time 4500. Intuitively, a build-order which maximizes for a specific time will start by producing worker units in order to gather more resources, and only start producing army units as the time limit approaches, which is evident by the long plateaus of army value in the green line. Strategically this may be a problem, since it may leave us vulnerable to attack during early stages of the

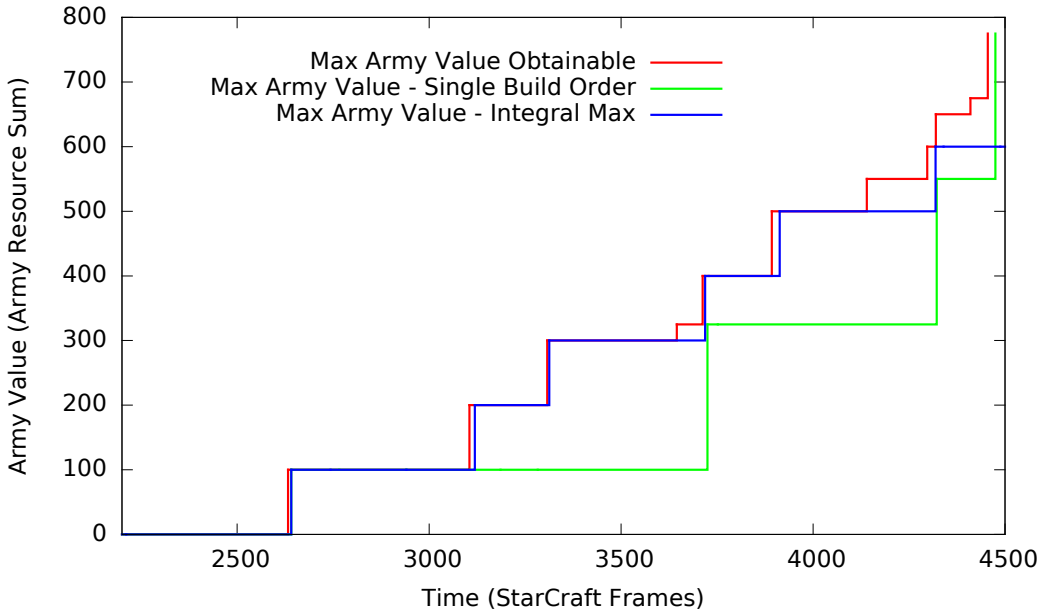


Figure 7.1: Shown are three lines which demonstrate the results of army value maximization build-order search, up to a maximim of 4500 STARCRAFT game frames. The red line is the maximum possible army value obtainable by any build-order at a given time. The green line is the army value at any given time for the single build-order which maximizes the army value at time 4500. The blue line is the army value for the single build-order which maximizes the area under the amy value curve.

build-order when we have not yet produced any army units. To illustrate this, the red line plots the maximum possible army value obtainable by *any* build-order for a given time. We see that at time 3300 and 4000 there are significant gaps between the maximum obtainable army value and the value obtained by the build-order in the green line, leaving us vulnerable at those times.

In order to find a less exploitable build-order which still produces a large amy value for a given time, we propose a method which doesn't maximize the army value at a given time, but instead maximizes the area of the army value curve up to a given time. The blue line in Fig. 7.1 represents the army value at any given time for the single build-order which maximizes the area under the army value curve up to time 4500. We can see that while this build-order does not produce an army value as high as the green line, it is much less exploitable to the maximum obtainable army value at any given point during

the build-order, as there are no significant gaps between the blue and red lines in the graph. Given these promising initial results, we plan to investigate this method of integral maximizing build-order search in the future by using SparCraft combat simulations in place of this simpler army value evaluation.

7.2.2 Improved Combat Simulation

In chapter 4 we discussed SparCraft, our STARCRAFT combat simulation system, and in section 4.7 we detailed the issues which arise when integrating combat search algorithms into the STARCRAFT game engine. We feel that one important area for future research which will greatly improve integration results is to improve the combat simulation so that it is closer to that of the STARCRAFT game engine. One of the main issues that arose in section 4.7 was that the timing of attack and movement cooldowns was often slightly different in the simularot and the real game. By improving SparCraft to more closely resemble STARCRAFT, then it is possible that the actions produced by the simulator will be more easily integreated into the actual game engine. Some initial investigation into improving the movement simulation of SparCraft was done by Schneider and Buro in [71], which showed that STARCRAFT movement mechanics such as acceleration and turning which are not modeled by SparCraft caused cumulative errors which made the SparCraft and STARCRAFT states diverge rapidly over time. We would like to pursue this investigation further and incorporate the results back into our simulation package.

7.2.3 Machine Learning State Evaluations

The search algorithms for RTS combat in chapter 4 and the HPS algorithm in chapter 5 both make extensive use of game playouts for state evaluation in their tree search. While these playouts proved to be far more accurate than simple formula-based evaluations and produced much stronger results they are still quite computationally slow, taking up to a thousand times longer than a formula-based evaluation. We feel that these search algorithms could be improved dramatically if some sort of machine learned state evaluation could replace these game playouts. The work of Erickson and Buro in [32] showed

promising initial results that machine learning techniques can be used to learn a global state evaluation for STARCRAFT, and so we believe that this technique could also be used for learning an RTS combat evaluation function. Also, with the recent success of deep neural networks in Google DeepMind's AlphaGo program [72], we feel that there could be significant advancements made in RTS AI by utilizing deep neural networks for areas such as state evaluation.

Bibliography

[1] David W. Aha, Matthew Molineaux, and Marc J. V. Ponsen. Learning to win: Case-based plan selection in a real-time strategy game. In *ICCBR*, pages 5–20, 2005.

[2] Phillipa Avery, Sushil Louis, and Benjamin Avery. Evolving coordinated spatial tactics for autonomous entities using influence maps. In *Proceedings of the 5th international conference on Computational Intelligence and Games*, CIG'09, pages 341–348, Piscataway, NJ, USA, 2009. IEEE Press.

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

[4] Nicolas A Barriga, Marius Stanescu, and Michael Buro. Building placement optimization in real-time strategy games. In *Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2014.

[5] Nicolas A Barriga, Marius Stanescu, and Michael Buro. Puppet search: Enhancing scripted behavior by look-ahead search with applications to real-time strategy games. In *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2015.

[6] BioTools. Poker Academy - your source for great Poker software. `http://www.poker-academy.com/`, 2013.

[7] Filip Bober. TerranUAB. `https://github.com/filipbober/scaiCode/`, 2015.

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

[9] Augusto A.B. Branquinho and Carlos R. Lopes. Planning for resource production in real-time strategy games based on partial order planning, search and learning. In *Systems Man and Cybernetics (SMC), 2010 IEEE International Conference on*, pages 4205–4211. IEEE, 2010.

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

[11] Michael Buro. 2006 ORTS RTS game AI competition. `https://skatgame.net/mburo/orts/AIIDE06/index.html`, 2006.

[12] Michael Buro and Timothy Furtak. On the development of a free RTS game engine. In *GameOn Conference*, pages 23–27. Citeseer, 2005.

[13] Michael Buro and Alexander Kovarsky. Concurrent action selection with shared fluents. In *AAAI Vancouver, Canada*, 2007.

[14] Martin Certicky and Michal Certicky. Case-based reasoning for army compositions in real-time strategy games. In *Proceedings of Scientific Conference of Young Researchers*, pages 70–73, 2013.

[15] Michal Certicky. [SSCAI] student starcraft AI tournament 2013. `http://www.sscaitournament.com/`.

[16] Michal Certicky. Student StarCraft AI Tournament. `http://sscaitournament.com/`, 2015.

[17] H. Chan, A. Fern, S. Ray, N. Wilson, and C. Ventura. Extending online planning for resource production in real-time strategy games with search. *ICAPS Workshop on Planning in Games*, 2007.

[18] H. Chan, A. Fern, S. Ray, N. Wilson, and C. Ventura. Online planning for resource production in real-time strategy games. In *Proceedings of the International Conference on Automated Planning and Scheduling, Providence, Rhode Island*, 2007.

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

[20] David Churchill. Build-Order Search System. `https://github.com/davechurchill/ualbertabot/tree/master/BOSS`, 2016.

[21] David Churchill. SparCraft: Open Source StarCraft Combat Simulation. `https://github.com/davechurchill/ualbertabot/wiki/SparCraft-Home`, 2016.

[22] David Churchill. UAlbertaBot. `https://github.com/davechurchill/ualbertabot/`, 2016.

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

[24] 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.

[25] David Churchill and Michael Buro. Hierarchical portfolio search: Prismata's robust ai architecture for games with large search spaces. In *Proceedings of the Artificial Intelligence in Interactive Digital Entertainment Conference*, 2015.

[26] David Churchill, Abdallah Saffidine, and Michael Buro. Fast heuristic search for RTS game combat scenarios. In *AI and Interactive Digital Entertainment Conference, AIIDE (AAAI)*, 2012.

[27] Rémi Coulom. Efficient selectivity and back-up operators in Monte-Carlo tree search. In *Proceedings of the 5th Conference on Computers and Games (CG'2006)*, volume 4630 of *LNCS*, pages 72–83, Torino, Italy, 2006. Springer.

[28] Holger Danielsiek, Raphael Stuer, Andreas Thom, Nicola Beume, Boris Naujoks, and Mike Preuss. Intelligent moving of groups in real-time strategy games. *2008 IEEE Symposium On Computational Intelligence and Games*, pages 71–78, 2008.

[29] Douglas Demyen and Michael Buro. Efficient triangulation-based pathfinding. *Proceedings of the 21st national conference on Artificial intelligence - Volume 1*, pages 942–947, 2006.

[30] 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.

[31] EmscriptenProject. emscripten. `http://emscripten.org/`, 2014.

[32] Graham Kurtis Stephen Erickson and Michael Buro. Global state evaluation in starcraft. In *AIIDE*, 2014.

[33] Kenneth D. Forbus, James V. Mahoney, and Kevin Dill. How qualitative spatial reasoning can improve strategy game AIs. *IEEE Intelligent Systems*, 17:25–30, July 2002.

[34] 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.

[35] GGBeyond. e-Sports earnings. `http://www.esportsearnings.com/`, 2013.

[36] Johan Hagelbäck. Potential-field based navigation in StarCraft. In *CIG (IEEE)*, 2012.

[37] Johan Hagelbäck and Stefan J. Johansson. Dealing with fog of war in a real time strategy game environment. In *CIG (IEEE)*, pages 55–62, 2008.

[38] Johan Hagelbäck and Stefan J. Johansson. A multiagent potential field-based bot for real-time strategy games. *Int. J. Comput. Games Technol.*, 2009:4:1–4:10, January 2009.

[39] Adam Heinermann. Broodwar API. `https://github.com/bwapi/bwapi`, 2013.

[40] Stephen Hladky and Vadim Bulitko. An evaluation of models for predicting opponent positions in first-person shooter video games. In *CIG (IEEE)*, 2008.

[41] Hai Hoang, Stephen Lee-Urban, and Héctor Muñoz-Avila. Hierarchical plan representations for encoding strategic game AI. In *AIIDE*, pages 63–68, 2005.

[42] Glenn Iba. A heuristic approach to the discovery of macro-operators. *Machine Learning*, 3(4):285–317, 1989.

[43] U. Jaidee and H. Muñoz-Avila. CLASSQ-L: A Q-learning algorithm for adversarial real-time strategy games. In *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2012.

[44] Niels Justesen, Bryan Tillman, Julian Togelius, and Sebastian Risi. Script-and cluster-based uct for starcraft. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, pages 1–8. IEEE, 2014.

[45] Froduald Kabanza, Philipe Bellefeuille, Francis Bisson, Abder Rezak Benaskeur, and Hengameh Irandoust. Opponent behaviour recognition for real-time strategy games. In *AAAI Workshops*, 2010.

[46] Levente Kocsis and Csaba Szepesvari. Bandit based Monte-Carlo planning. In *Proceedings of the European Conference on Machine Learning*, pages 282–293, 2006.

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

[48] Alexander Kovarsky and Michael Buro. A first look at build-order optimization in real-time strategy games. In *Proceedings of the GameOn Conference*, pages 18–22, 2006.

[49] LunarchStudios. Prismata. `http://www.prismata.net/`, 2015.

[50] Charles Madeira, Vincent Corruble, and Geber Ramalho. Designing a reinforcement learning-based adaptive AI for large-scale strategy games. In *AI and Interactive Digital Entertainment Conference, AIIDE (AAAI)*, 2006.

[51] Bjorn Mattsson. Odin. `http://plankter.se/projects/odin/`, 2015.

[52] Christopher Miles and Sushil J Louis. Co-evolving real-time strategy game playing influence map trees with genetic algorithms. In *Proceedings of the International Congress on Evolutionary Computation, Portland, Oregon*, 2006.

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

[54] Adam Montgomerie. MooseBot. `https://github.com/iarfmoose/MooseBot`, 2014.

[55] John Forbes Nash. Equilibrium points in n-person games. *Proceedings of the National Academy of Sciences of the United States of America*, 36(1):48–49, 1950.

[56] Santiago Ontañón. microRTS. `https://github.com/santiontanon/microrts`, 2016.

[57] 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*, pages 1652–1658. AAAI Press, 2015.

[58] 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.

[59] Santiago Ontanón, 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*, 2013.

[60] Jeff Orkin. Three states and a plan: The A.I. of F.E.A.R. In *GDC*, 2006.

[61] Nasri Othman, James Decraene, Wentong Cai, Nan Hu, and Alexandre Gouaillard. Simulation-based optimization of StarCraft tactical AI through evolutionary computation. In *CIG (IEEE)*, 2012.

[62] 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.

[63] Marc Ponsen and Pieter Spronck. Improving adaptive game AI with evolutionary learning. In *University of Wolverhampton*, pages 389–396, 2004.

[64] Craig W. Reynolds. Steering behaviors for autonomous characters. *Proceedings of Game Developers Conference 1999*, pages 763–782, 1999.

[65] Florian Richoux, Alberto Uriarte, and Santiago Ontañón. Walling in strategy games via constraint optimization. In *AIIDE*, 2014.

[66] Martin Rooijackers. Letabot. `http://wiki.teamliquid.net/starcraft/LetaBot`, 2015.

[67] Abdallah Saffidine, Hilmar Finnsson, and Michael Buro. Alpha-Beta pruning for games with simultaneous moves. In *Proceedings of the Twenty-Sixth Conference on Artificial Intelligence*, July 2012.

[68] 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.

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

[70] Jonathan Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 11(11):1203–1212, 1989.

[71] Douglas Schneider and Michael Buro. Starcraft unit motion: Analysis and search enhancements. In *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2015.

[72] 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.

[73] Greg Smith, Phillipa Avery, Ramona Houmanfar, and Sushil Louis. Using co-evolved RTS opponents to teach spatial tactics. In *CIG (IEEE)*, 2010.

[74] Marius Stanescu, Nicolas 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)*, 2015.

[75] Marius Stanescu, Nicolas A Barriga, and Michael Buro. Hierarchical adversarial search applied to real-time strategy games. In *AIIDE*, 2014.

[76] Marius Stanescu, Sergio Poo Hernandez, Graham Erickson, Russel Greiner, and Michael Buro. Predicting army combat outcomes in starcraft. In *AIIDE*. Citeseer, 2013.

[77] M. Stolle and D. Precup. Learning options in reinforcement learning. *Abstraction, Reformulation, and Approximation*, pages 212–223, 2002.

[78] Nathan R Sturtevant. Benchmarks for grid-based pathfinding. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(2):144–148, 2012.

[79] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. The MIT Press, March 1998.

[80] Gabriel Synnaeve and Pierre Bessiere. 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, 2011.

[81] 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.

[82] Gabriel Synnaeve and Pierre Bessiere. Special tactics: a Bayesian approach to tactical decision-making. In *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, pages 409–416, 2012.

[83] Adrien Treuille, Seth Cooper, and Zoran Popović. Continuum crowds. *ACM Transactions on Graphics*, 25(3):1160–1168, 2006.

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

[85] Alberto Uriarte and Santiago Ontañón. Automatic learning of combat models for rts games. In *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2015.

[86] Ben G. Weber, Michael Mateas, and Arnav Jhala. Applying goal-driven autonomy to StarCraft. In *Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 2010.

[87] 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.

[88] Ben G. Weber, Peter Mawhorter, Michael Mateas, and Arnav Jhala. Reactive planning idioms for multi-scale game AI. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, pages 115–122, 2010.

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

[90] Andrew R. Wilson. Masters of war: History's greatest strategic thinkers. `http://www.thegreatcourses.com/courses/masters-of-war-history-s-greatest-strategic-thinkers.html`, 2012.

[91] Sijia Xu. Overkill. `https://github.com/sijiaxu/Overkill`, 2015.

[92] Jay Young and Nick Hawes. Evolutionary learning of goal priorities in a real-time strategy game. In *AIIDE*, 2012.

[93] Gu Zhan. NUSBot. `https://code.google.com/archive/p/nus-bot/`, 2014.