Adversarial Search and Spatial Reasoning

in Real Time Strategy Games

A Thesis
Submitted to the Faculty
of
Drexel University
by
Alberto Uriarte
in partial fulfillment of the
requirements for the degree
of
Doctor of Philosophy
May 2017

Acknowledgments

I would like to thank my advisor Santiago Ontañón for his excellent guidance and support throughout my dissertation. I would also like to thank Florian Richoux to share with me his passion about constraint satisfaction/optimization problems; David Churchill for our interesting discussions, organizing the StarCraft AI competition and his great tools SparCraft and BOSS; Adam Heinermann for maintaining BWAPI that makes possible to do research in StarCraft; and the always helpful StarCraft AI bot community.

I would also like to thank all my committee members, Marcello Balduccini, Christopher Geib, Michael Buro and Gabriel Synnaeve, for their invaluable comments and discussions to improve my work. I would also like to thank all the members of the Drexel AI and Games lab. Josep Valls-Vargas, Sam Snodgrass and Brandon Packard, thank you for creating a great working environment and for your constant feedback. I want to thank Alex Duff, Ehsan B Khosroshahi and Bander Alsulami for their encouragement, moral support and proofreading my work.

Table of Contents

List of Tables

List of Figures

Abstract

Adversarial Search and Spatial Reasoning
in Real Time Strategy Games
Alberto Uriarte
Santiago Ontañón, Ph.D.

For many years, *Chess* was the standard game to test new Artificial Intelligence (AI) algorithms for achieving robust game-playing agents capable of defeating the best human players. Nowadays, games like *Go* or *Poker* are used since they offer new challenges like larger state spaces, or non-determinism. Among these testbed games, Real-Time Strategy (RTS) games have raised as one of the most challenging. The unique properties of RTS games (simultaneous and durative actions, large state spaces, partial observability) make them a perfect scenario to test algorithms able to make decisions in dynamic and complex situations. This thesis makes a contribution towards achieving human-level AI in these complex games. Specifically, I focus on the problems of performing adversarial search in domains (1) with extremely large decision and state spaces, (2) where no forward model is available, and (3) the game state is partially observable. Additionally, I also study how spatial reasoning can be used to reduce the search space and to improve the RTS playing bots.

Chapter 1: Introduction

Real-time Strategy (RTS) games pose a significant challenge to Artificial Intelligence (AI), for which we do not currently have a good solution, as evidenced by the fact that humans are still better than "non cheating" AIs. By non cheating AIs we mean an AI that does not have access to the full game state (full observability) and unlimited resources. In these kinds of games, players have to control resources, produce units to fight, and defeat an opponent, and do so under real-time and partial observability constraints. Many aspects of the game mechanics of RTS games make the reasoning required to play these games very complex and it is hard to achieve good game play performance with the current state-of-the-art AI algorithms. Most of the commercial RTS games use handcrafted scripted methods that are easy to defeat for human players and hard to create and maintain by the game designers. The biggest problem of a scripted AI is the lack of adaptation and reaction. Human players can easily detect patterns and weaknesses in order to exploit them systematically. This leads to a loss of interest from the player to keep playing the game because it does not present any challenge anymore.

Creating good game-playing agents for RTS games is hard mainly because of the following reasons: RTS games are partially observable, the game state is constantly changing (real-time), and players have to control a large set of units with different characteristics, which results in large state spaces and an enormous set of possible actions to consider at each instant of time. Additionally, in some RTS games the actions are stochastic (the same input produces different outputs). All of these problems will be elaborated in detail below. Because of these reasons, existing search, planning, or adversarial planning techniques do not scale well in these domains (not even to the most trivial RTS game scenarios).

Having algorithms that can play RTS games, in addition to being a hard and unsolved problem, might have many applications beyond games. On the one hand, having standard game-playing algorithms for RTS games will be a significant contribution to both the game industry (who will be able to create more appealing RTS games, and also do so in an easier way). On the other hand, these algorithms would be useful in domains requiring planning and coordination of multiple units in real time, such as disaster recovery or exploration missions. In general, there are many subproblems in RTS games for which we do not yet have good solutions; thus, solving these problems will be a significant step forward in AI, and can expand the problem solving capabilities of current algorithms.

This document is organized as follows: Section 1.1 introduces Real-Time Strategy (RTS) games and their challenges from an AI point of view. Chapter 2 reviews the current state of the art towards addressing each of these challenges. Chapter 3 presents our contributions to adversarial search for RTS games, while Chapter 4 presents three new algorithms for spatial reasoning problems (terrain analysis, walling and kiting). Finally, Chapter 5 describes an AI agent (or bot), called Nova, that can play StarCraft (a popular RTS game), and where all the aforementioned contributions were implemented for evaluation; along with how to test the performance of StarCraft bots.

1.1 Real Time Strategy Games

Real-time Strategy (RTS) is a game genre where players need to build an economy (gathering resources and building facilities) and a military power (training units and researching technologies) in order to defeat their opponents (destroying their army and base). From a theoretical point of view, the main differences between RTS games and traditional turn-based board games such as Chess are:

- Players can issue actions at the same time (**simultaneous moves**), and most actions take some amount of time to complete (**durative actions**). While, in Chess, moves alternate between players and the actions are instantaneous.

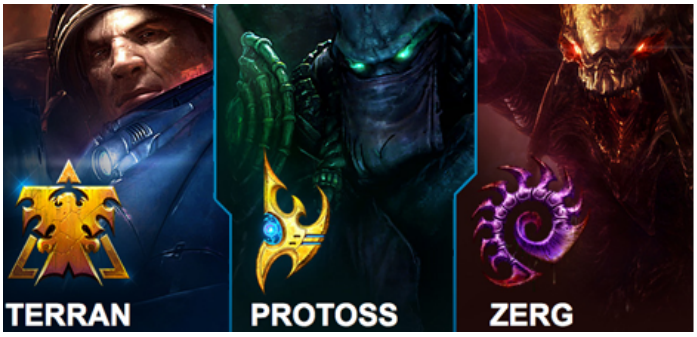


Figure 1.1: Screenshot of race selection in StarCraft.

- RTS games are **real-time**, which means players do not need to wait for their opponents to issue the next move, for example, StarCraft is updated at 24 iterations per second, hence players can issue a new move every 42 ms. In Chess, players may have several minutes to decide the next move.
- Most RTS games are **partially observable** due the *fog-of-war* that only lets observe the area of the map under the sight range of player's units, while games like Chess are perfect information games.
- Most RTS games are **non-deterministic**, since some actions might have a chance of failure, and some other actions might have multiple possible outcomes.
- They have a **larger state space** (number of possible game states) and a **large action space** (number of possible actions given a game state) than classical board games such as Chess or Go.

The next section describes the two RTS games that I have used for evaluation in this work: (1) StarCraft, one of the most popular RTS games of all time and which has become a standard benchmark for RTS game AI research in the past few years, and (2) μRTS, a minimalistic RTS game designed specifically for AI research.

1.1.1 StarCraft

StarCraft: Brood War is an immensely popular RTS game released in 1998 by Blizzard Entertainment. StarCraft is set in a science fiction based universe where the player must choose one of the three races: Terran, Protoss or Zerg (Figure 1.1). One of the most remarkable aspects of StarCraft is that the three races are extremely well balanced:

- **Terrans** provide units that are versatile and flexible giving a balanced option between *Protoss* and *Zergs*.
- **Protoss** units have lengthy and expensive manufacturing process, but they are strong and resistant. These conditions make players follow a strategy of quality over quantity.
- **Zergs**, the insectoid race, units are cheap and weak. They can be produced fast, encouraging players to overwhelm their opponents with sheer numbers.

StarCraft is a war-like game where we have two types of resources (minerals and Vespene gas) to gather. The stored resources can be spent in (1) building facilities (such as barracks) to unlock new types of units; (2) researching new technologies to unlock new abilities or upgrade the units; or (3) training new units (such as tanks or marines) to have a stronger army. There are two strategic ways to spend the resources: one is training more workers and building more bases near resources in order to increase the gathering income and improve your *economy*; the other is training more military

Figure 1.2: A screenshot of StarCraft: Brood War.

units or improving them doing research to increase your *military power*. The first one is a long-term investment (with more resources you can train a bigger army later) while building an army is a short-term investment (players need units to attack or defend themselves). In StarCraft there is a third type of resources that limits the amount of units a player can have: supply. Supply can be increased by building certain facilities or units, but there is a 100 cap that players cannot exceed. Every living unit blocks an amount of supplies (it ranges from 0.5 to 16), hence StarCraft has a maximum limit of 200 units per player. While there is not a limit for the number of buildings, they are limited by the space available in the map. Once players have an army, they can command the units to perform different tasks such as scouting, attack or defense. These require quick and reactive control of each of the units to maneuver them in order to succeed in battle. Figure 1.2 shows a screenshot of StarCraft where a player controls the Terran race.

The board, called "map" in StarCraft, can change from game to game. A typical StarCraft map is defined as a rectangular grid, where the $width \times height$ of the map is measured in "build tiles", which are squares of 32×32 pixels. Moreover, path-finding in StarCraft occurs at a finer resolution of "walk tiles", which are squares of 8×8 pixels. Map dimensions can range from 64×64 to 256×256 build tiles and tiles can have different properties such as altitude.

In Appendix A we explain in more detail all StarCraft concepts that we will use in this dissertation, like common strategies such as rush or harass, concepts such as build-order, and all the relevant properties of StarCraft units.

1.1.2 μRTS

μRTS [1] is an open source RTS game specifically designed for research [2]. It is highly configurable, allowing researchers to enable or disable RTS features such as partial observability or non deterministic actions. One of the advantages of using μRTS is the access to the game forward model (something not available in commercial games such as StarCraft) and the possibility of customizing the units and maps. μRTS allows the creation of controlled scenarios and also allows speeding up experimentation by not requiring to run the graphical interface when running experiments..

Maps are simple rectangular grids without any size limitation (they can be very small or very large), there is only one type of resources and all the units have the same size: 1 cell. By default μRTS has a set of ground units available, for more details see Appendix B

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

Table 1.1: State space complexity of different games.

Game	State Space
Tic-tac-toe	10^3
Limit Texas Hold'em Poker	10^{13} [3]
Reversi (Othello)	10^{28}
Chess	10^{47} [4]
Go	10^{171} [5]
μRTS	10^{110}
StarCraft	$> 10^{2791}$

1.1.3 Complexity of RTS Games

From a theoretical point of view, there are two different ways to compute the complexity of games like StarCraft. One is the **game complexity** measured as the *state space complexity* and the *game tree complexity*. The other is the **computational complexity**, concerning classifying the problem of determining a wining strategy for a given RTS game situation, into one of the well known complexity classes (such as P or NP), and determining whether the problem is decidable or not.

Game Complexity

The **state space complexity** is the number of legal game positions reachable from a game state. In some games we can easily compute this number, but when this is not feasible, we can attempt to compute an upper bound (that might include illegal positions or configurations that are impossible to reach), or a lower bound. In StarCraft, one possible estimation is to consider all possible configurations of the current units; more specifically we can compute it as:

$$\prod_{j} (numValidPositions(j) \times dynamicProperties(j))$$

where j is a StarCraft unit in the current game state, $numValidPositions(j)$ is the number of valid positions of unit j given a map, and $dynamicProperties(j)$ is the number of different states in which a unit can be in, based on the unit properties that can change during a game (like hit points, energy or shield). The equation can be generalized to consider an average type of unit in a specific map. For example, just considering the unit hit points (and ignoring other properties), the average maximum hit points of a Terran ground unit is 76; and the walkable tiles of a common map like Benzene (151322 walkable tiles). This results in $(76 \times 151322)^n$, where n is the total number of units in the current game state. Using the upper bound of the maximum number of units possible in a StarCraft game (400), gives us the number of $9533286^{400} \approx 10^{2791}$. If we add other factors playing a role in the game, we can obtain even larger numbers. In the case of μRTS, if we consider a map with size 12x12 (144 tiles) and only *Light* units (which have at most 4 hit points), an initial estimation of the number of possible states is $(4 \times 144)^n$, where this time the number of possible units (n) is limited by the number of resources; typically in a μRTS map of 12x12 there are 80 minerals, allowing us to produce at most 40 *Light* units. These numbers give us an approximate estimation of $576^{40} \approx 10^{110}$. Table 1.1 shows a comparison of the state space complexity for different games.

The **game tree complexity** can be computed as b^d, where b is the average branching factor of the game tree and d the depth of the game. In classical board games players can issue only one action per turn, but in StarCraft the AI can issue actions simultaneously to any number of units in the current game state. Then, one way to estimate the branching factor in StarCraft is by the following equation: $actions^{units}$, where *actions* is the average number of actions each unit can execute, and *units* is the average number of units in the current game state. Let us make the following assumptions: (1) in StarCraft the maximum number of units is 100 (200 if playing Zerg race),

Table 1.2: Game tree complexity of different games.

Metric	**Chess**	**Go**	**μRTS**	**StarCraft**
Branching Factor (b)	36	180	10^{14}	10^{75}
Depth of the Game (d)	80	200	3000	36000
Effective Branching Factor (b^*)	1.7[†]	30 [7]	unknown	unknown

so lets assume an average of 75 units; (2) units can execute between 1 (for instance *Supply Depots* are buildings whose only action is to be "idle") to 43 actions (like moving to any of the 8 cardinal directions, attacking enemy units in attack range, using a special ability or building a new facility), taking into account that some actions have cool-down times and thus they will have temporarily only one action available ("cancel" current action), we can assume an average of 10 possible actions. This results in a conservative estimate for the branching factor $b = 10^{75}$, only considering units (ignoring the actions buildings can execute). Now, to compute d, we simply consider the fact that typical StarCraft games last for about 25 minutes, which results in $d \approx 36000$ (25 minutes $\times$ 60 seconds $\times$ 24 frames per second). In the μRTS case, let us consider a 12x12 map where we can produce at most 40 *Light* units or 80 *Worker* units. Let us assume that the average number of units is 20 and the average number of actions a unit can perform is 5 (4 move/attack directions and idle). This gives us a branching factor of $5^{20} \approx 10^{14}$. Usually a μRTS game is considered a tie if it reaches 3000 frames (previous work has reported branching factors in 12x12 maps in μRTS of up to 10^{19} [6]). These numbers are summarized in Table 1.2, showing the significant difference in complexity from traditional games to RTS games.

Moreover, different algorithms exist that can prune game trees in order to reduce the number of branches to expand. The *effective branching factor* (b^*) [8] takes this into account by computing the branching factor that a uniform tree of depth d (equal to the length of the solution found) would have in order to contain a total number of nodes T (equal to the nodes generated during the search using a game tree search algorithm that can prune branches). Specifically, the effective branching factor b^*, can be computed by solving the following equation:

$$T = \sum_{i=1}^{d} (b^*)^i = \frac{b^*((b^*)^d - 1)}{b^* - 1} \tag{1.1}$$

When b^* is sufficiently large, we can estimate it by $b^* = T^{(1/d)}$, otherwise, numerical solutions to the previous equations exist. The effective branching factor is a good metric to measure the quality of a tree pruning strategy. A better pruning strategy will decrease the effective branching factor. For example, the effective branching for the *minimax* game tree search algorithm is equal to the average branching factor; while in *alpha-beta* pruning with a good move ordering we can reach an effective branching factor of the square root of the average branching factor.

Computational Complexity

The computational complexity of a game tries to quantify the amount of resources needed (execution time and memory space) to solve computational problems regardless of the algorithm used. Using mathematical models of computation we can classify the complexity of a game in different classes. Games can be classified by the number of players, the number of moves that can be made or the level of information (perfect or imperfect information). If a game has restrictions on the number of moves (length of the game), we call it a bounded-length game; otherwise it is an unbounded-length game. Games where the moves are reversible[2] are usually unbounded-length games since we can keep doing-redoing the moves indefinitely. Two-player and bounded-length games tend to be

[†] `http://www.talkchess.com/forum/viewtopic.php?t=48281`

[2] Keep in mind that when we consider a generic game played in an N×N board, rules to avoid reversible moves like *50 moves* for chess or *Ko* for Go are not applied.

PSPACE-complete like Reversi [9], while some two-player and unbounded-length games like Chess or Go, have been proved to be **EXPTIME**-complete [10, 11].

Chandra, Kozen and Stockmeyer [12] defined the concept of *alternating Turing machine* (ATM) were the transitions alternate between two different sets of states (that we can see as two players). Reif [13] described two new ATMs: *private alternating Turing machine* (PATM) where some of the actions of one of the players can be observed; and *blind alternating Turing machine* (BATM) where one of the players cannot see information about the other. In two-player unbounded-length games, if players have perfect information (ATM), they are **EXPTIME**, if there is private information (PATM) they are **2-EXPTIME** and if there is blindfold information (BATM) they are **EXPSPACE**.

Now if we consider some of the features of RTS games (they are two-player games with imperfect information due to the fog-of-war and simultaneous moves), we can guess that RTS game complexity is at least **2-EXPTIME**, but no formal proof yet exists. The only work on specifically determining the computational complexity of RTS games is Viglietta's work [14]. Viglietta defined several meta-theorems. One of them states that any game exhibiting both location traversal (location that the player must traverse) and single-use paths is at least **NP**-hard. To prove his statement, he showed a reduction from Hamiltonian paths, and pointed how these two conditions are present in RTS games like StarCraft.

Another approach is considering only a subset of full RTS games. In that case, Furtak and Buro [15] represented a small-combat situation as a graph game, more specifically, as an attrition game. An attrition game (AG) is a graph-based simultaneous move game in which two players, black and white, attempt to destroy each other's nodes. A player is said to win if she destroys all opposing nodes while preserving at least one of her nodes. They proved, using a reduction from a QBF game, that computing the existence of deterministic winning strategies for the basic attrition game in general is **PSPACE**-hard and **EXPTIME**.

Both the work of Viglietta and of Furtak and Buro, provide lower bounds on the computational complexity of RTS games. However, it is still to be determined whether they are **2-EXPTIME** or not, or even whether they belong to a higher complexity class or not.

1.2 Challenges in RTS Game AI

All the factors presented in the previous section make StarCraft a significant challenge for artificial intelligence (as evidenced by the fact that humans have always been able to defeat the best AIs available for StarCraft in the yearly man-vs-machine match at the AIIDE StarCraft AI competition[3]). Early research in AI for RTS games [16] identified the following six challenges:

1. Resource management
2. Decision making under uncertainty
3. Spatial and temporal reasoning
4. Collaboration (between multiple AIs)
5. Opponent modeling and learning
6. Adversarial real-time planning

Recent research has identified several additional challenges, such as how to exploit the large amount of game replays from professional human players. Below, we describe current challenges in RTS Game AI, grouped in five main different areas (similar with the areas presented in our journal article [17]).

Adversarial Planning in Large Real-Time Domains

As mentioned in the previous section, the sizes of the state space and the action space are much larger than traditional board games like Chess or Go. Therefore, standard adversarial game tree search approaches are not directly applicable due the large game tree to explore. One possible solution

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

is using multiple levels of abstraction, and perform an adversarial game tree search in specific sub-problems, such as long-term planning (like the dilemma between improving the economy or investing in a bigger army), or short-term planning (like military unit coordination to exploit the terrain or enemy weaknesses). Since these abstractions are based on time, the temporal reasoning is an important element. *When* to execute a plan is as important as *what* plan you want to execute. Some examples of plans that are time sensitive and their effects are not immediate are: attacks and retreats to gain an advantage, spend resources into research or building construction, and strategy switching. Moreover, we may face the problem of the lack of a forward model in the level of representation we are working. For example, we do not have access to the source code of commercial games (like StarCraft), therefore we have to create our own forward model. Or in the case of a high level representation, we need to somehow model the combat between large armies.

Learning

Learning methods have received some attention in order to solve some RTS sub-problems. There are three different types of learning problems that can be addressed in RTS games (Chapter 2 reviews existing work on these):

- **Prior Learning**: A type of *offline learning* that usually uses large amount of data, such as game replays, to infer knowledge beforehand. An example would be to learn all possible build orders in order to use the most effective one in a given game. These technique have been applied in traditional board games such as Chess, where researchers used opening books or end-game tables to improve the evaluation function. In recent years there has been some initial work on how to exploit the large dataset of human expert game replays for military decisions, following the ideas in AlphaGo [18], but it remains unclear how to exploit them farther than for combat scenarios.
- **In-Game Learning**: Also known as *online learning*, it involves improving the strategy while playing the game. The idea is to adapt to the actions the current opponent tends to perform (opponent modeling). Common techniques like reinforcement learning cannot be directly applied here due to the limited amount of data that can be gathered during a single game in comparison with the large state space that needs to be explored and also due to the partially observable features of RTS games.
- **Inter-Game Learning**: Another type of *offline learning* that uses the knowledge of the last games played to improve the next one. Some approaches used simple game-theoretical solutions to select the most promising strategy among a pool of predefined strategies. However, these approaches cannot be used to discover new strategies.

Spatial Reasoning

Any action where the terrain has an influence on the decision is called spatial reasoning. There are strategy actions like (1) building placement, which needs to consider the terrain particularities to avoid getting stuck in an area of the map, or be used as an advantage to close narrow passages; (2) how to decide where is the best location to build a new base and start gathering more resources; or (3) which is the safer path to reach a goal. Other actions are more tactical like (1) dynamic obstacle avoidance; (2) leading your opponent into an ambush; or (3) taking advantage of having own units in a higher ground than your enemy (for example, in StarCraft units have less weapon accuracy when they attack to units on a higher ground). Algorithms to detect an exploit these and more terrain properties are still needed.

Imperfect Information

Adversarial planning for imperfect information games in domains of the size of RTS games is still an open challenge. In some RTS games, **game states are partially observable**, and players cannot observe the whole game map (as is possible in Chess). Only the portion of the map that is under the

Table 1.3: Imperfect information game properties for different games.

	Poker	**Kriegspiel**	**μRTS**	StarCraft
Initial board configurations	1,081*	1	1	1-12†
Observable moves	YES	NO	YES	YES
Hidden moves	NO	YES	YES	YES
Exposing moves	NO	NO	YES	YES
Gathering moves	NO	YES	YES	YES

sight range of the units of a player is observable to that player, the rest of the map is covered under the *fog-of-war*. This uncertainty can be mitigated by scouting (sending units to explore territory currently under the fog-of-war), and knowledge representation (inferring what is possible given what has been seen). Partial observability also raises the possibility of playing the *deception game*: An expert player may let her opponent see some of her units on purpose to mislead her opponent about her planned strategy. If a player does a wrong assumption about the opponent's strategy (and the opponent knows what the player is assuming), the opponent will have a great advantage.

Typical partially observable RTS games involve the following concepts (Table 1.3 shows a comparison of imperfect information game properties for different games):

- The set of **initial board configurations** is small; this is because in most RTS games, such as StarCraft players know which map they are playing on, and the only non-observable information at the start is where did the opponent player start, out of a predefined set of possible start locations. For example, in StarCraft there is a fixed set of "start locations" in each map, and thus a player only needs to determine in which of those did the opponent start in. In μRTS each map defines a given start location for both players, and thus the initial game state is known to both players.
- Some moves are **observable**: are those that are fully observable to all players.
- Some moves are **hidden** (those that an opponent performs in an area of the map that is outside of the sight range of the player's units).
- **Exposing moves** occur when a player executes an action that reveals information to her opponent (like moving a unit out of the opponent's fog-of-war).
- **Information gathering moves** are those that make some opponent information observable.

Task Decomposition

Trying to deal with all the problems that RTS exhibit is very hard, for this reason most existing approaches try to decompose the problem of playing RTS games into a set of smaller sub-problems to be solved (semi-)independently. A common task decomposition is:

- **Strategy**: is the highest level of abstraction where players make long-term decisions. Strategies are planned to be completed farther in the future and they are re-planned every time we have new information about the opponent. Finding the best counter-strategy against a given opponent is key to success in RTS games.
- **Tactics**: are medium-term decisions in order to carry out a strategy. They usually involve how to maneuverer a group of units or how to place buildings.
- **Reactive control**: short-term decisions to execute the desired tactic. Reactive control is usually concerned with controlling single units. Some examples of reactive control are: target selection, attack and flee movement, patrol or harassing.

*Number of possible hands for a 2 player Texas hold'em game after the flop.
†Assuming a 2 player game on a map with between 2 to 4 starting locations.

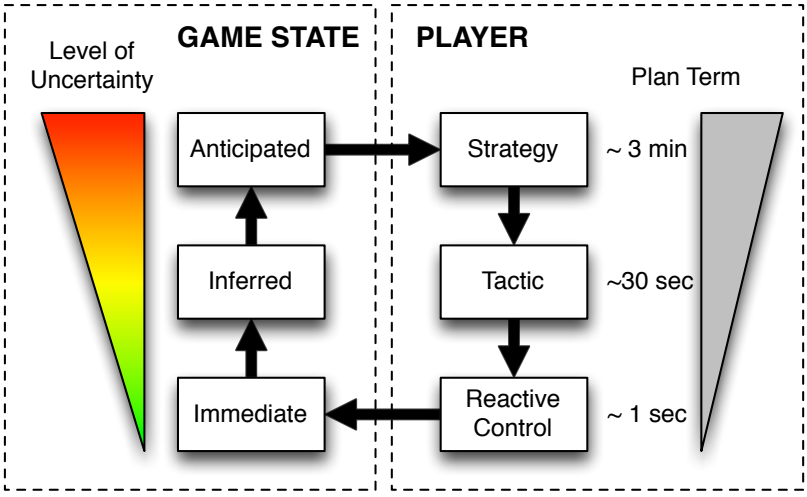


Figure 1.3: Different levels of abstraction in RTS games and how they are interconnected. The uncertainty increases for higher abstraction levels, while the plan term corresponds to a time estimation of task completion.

- **Terrain analysis**: how to decompose a given map into exploitable features such as regions, chokepoints, base locations or islands.
- **Imperfect information**: Because of the fog-of-war players do not have perfect information of the game state. Hence they have to perform information gathering actions like scouting in order to collect information about the opponent.

Figure 1.3 graphically illustrates how uncertainty can be categorized under *Game State* and temporal planning under *Player*. Usually, human players first conceive a global *strategy* consisting of a plan or set of plans towards a goal state that the player wants to achieve in the long term (several minutes in the case of StarCraft). A strategy can be decomposed into medium term plans called *Tactics*, those plans are localized, and affect only specific groups of units. Reactive control describes how the player controls individual units to maximize their efficiency in real-time. Once the actions of the units are executed, they have an *immediate* effect on the game state. These effects might be partial observables for other players, therefore intelligence gathering is needed to mitigate as much as possible the uncertainty about the game state. With all the observable game state gathered (including information coming from *Terrain analysis*), the player infers some of the non-directly observable properties with different degrees of confidence. Finally, with the inferred game state, the player is able to anticipate the future game state or the intentions of the opponent. To close the loop, this information will adjust the current strategy of the player. A similar idea was presented by Simon Dor in his "Heuristic Circle of Real-Time Strategy Process" [19].

For example, at the start of the game a player might decide to train an army to send it to attack the enemy as soon as possible (*rush* strategy); then during the attack the player can try to surround the enemy to cut any potential escape route (tactics) while every unit performs an *attack and flee* movement to minimize the casualties (reactive control). While the player is starting to execute the planned *rush* strategy, she sends a scout to the base of the enemy to discover that the opponent has built a second base (immediate knowledge). That might indicate that the opponent does not have an army (inferred knowledge) and he is looking for an economic advantage (anticipated). At this point, the player can decide to change the strategy and build a second base to avoid the economic disadvantage or keep with the initial strategy and harass the opponent.

1.3 Contributions

The work presented in this dissertation tackles some of the problems explained in the previous sections. More specifically, we present:

1. Techniques to scale up game tree search to handle adversarial planning in real-time domains. These techniques involve state and action abstraction techniques and learning Bayesian models from replays to inform MCTS search.

2. Forward models automatic learned from replay data that can perform similarly than hand crafted models; and extensions of well known combat models (Lanchester's Laws) to model combats between armies composed with different unit types.

3. A technique to deal with partial observability in RTS games based on maintaining a single *believe state*, which is an estimation of the actual game state given the information the AI has.

4. Spatial reasoning techniques that improve game state abstraction, and low-level action execution (walling, kiting).

5. An empirical evaluation of all of these contributions by incorporating them into a fully integrated bot called Nova that plays StarCraft. Along with a standard way to evaluate the performance of StarCraft bots.

Chapter 2: Background

As showed in Section 1.2, RTS games expose different challenges that RTS AIs need to address. This chapter reviews common algorithms to perform a game tree search that can be used to solve some of the challenges. We organize existing work using the same task decomposition presented above (Strategy, Tactics, Reactive control, Terrain Analysis and Intelligence Gathering) plus an extra section to Holistic approaches. For previous overviews the reader is referred to our journal article [17] or to Robertson and Watson [20] review.

2.1 Game Tree Search

A game tree is a directed graph whose nodes are game states and whose edges are actions that change the state of the game. The complete game tree for a given game is one where the root node is the initial position of the game, and that contains all possible moves from each position until the end of the game.

Game trees have been used extensively in game AI because they can be used to pick the best action in a given game state (for example using algorithms like *minimax* or its variants). However, not all games are suitable for this kind of algorithms. Lets first review how we can classify games [1]:

- Zero-sum: Whether the reward to all players sums to zero. That means if the players are in strict competition with each other (if one gains, another loses).
- Information: Whether players can fully observe the state of the game (*perfect information*) or some players can only observe part of the game state (*imperfect information*).
- Stochastic: Whether some actions have a chance to produce different states or they produce always the same outcome.
- Sequential: Whether the players apply actions sequentially (taking turns) or simultaneously.

We use the term *combinatorial game* to refer to a two-player game that is zero-sum, perfect information, deterministic and sequential [21]. In these kinds of games, *minimax* algorithms usually work pretty well. However, game trees for typical board games (like Chess or Go) are too large to be fully explored. For that reason, standard approaches based on *minimax* only explore the tree up to a certain depth. There are *minimax* variants that aim at reducing the part of the tree that needs to be explored by using, for example, pruning techniques. But for some games, like *Go*, even these enhancements are not sufficient to search the tree at the depth that it would be required to achieve good game play performance.

Having in mind the game tree complexity of RTS games, it is clear that game tree search in RTS games is an enormous challenge due to the large search space and branching factor. Additionally, there are some RTS game characteristics that make things even harder: RTS games are partially observable, can be stochastic (like StarCraft), they have simultaneous moves and they are "real-time" (the state of the game changes frequently, several times per second). Fortunately, some of these properties can be approximated with a new promising family of algorithms known as *Monte Carlo Tree Search* [1].

The main idea of *Monte Carlo Tree Search* (MCTS) is that the value of a state may be approximated using repeated stochastic simulations from the given state until a leaf node (or a terminal condition). Thus, rather than systematically constructing the whole tree as *minimax* algorithms do, MCTS algorithms only partially construct this tree, via sampling. Then MCTS algorithms (Algorithm 1) build a partial game tree in an incremental and asymmetric manner. At each iteration, the algorithm selects the best (or more promising) node of the current state using a *tree policy*.

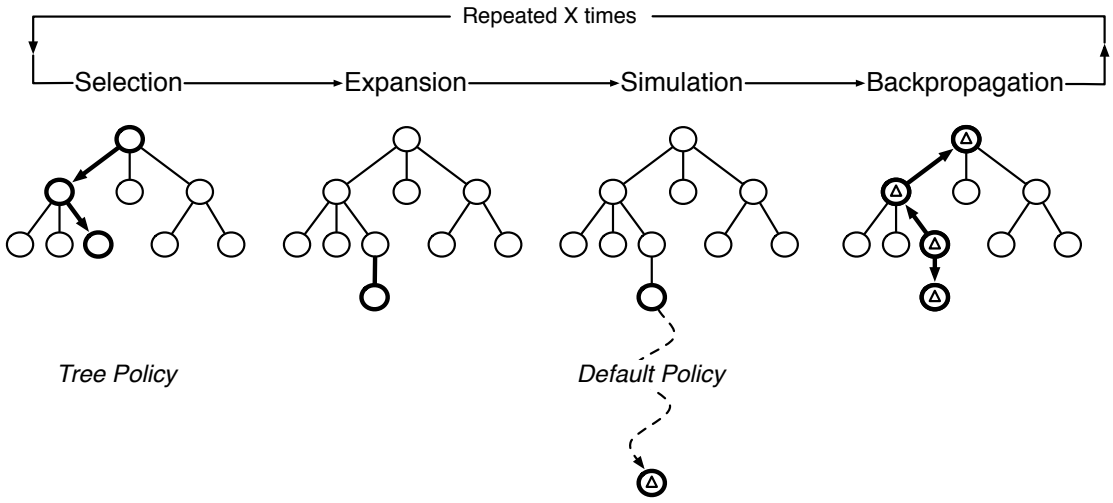


Figure 2.1: One iteration of the general MCTS approach (adapted from [1]).

The *tree policy* is used to balance the *exploration* (look in areas that have not been explored yet) and *exploitation* (look at the most promising areas of the tree). Then, a simulation is run from the selected node, using a *default policy* to know which moves to choose during the simulation, until a terminal node is reached. The simplest default policy that can be used to run the simulations is selecting uniform random moves. One of the benefits of MCTS is that during the simulation we do not have to evaluate the intermediate state. It is sufficient to evaluate the terminal state at the end of the simulation and backpropagate the value to the selected node. Figure 2.1 illustrates the steps of MCTS. As we can see in the previous explanation, the key points of MCTS, and the reason of the existence of so many different MCTS flavors, is how to define the *tree policy* (what child to chose) and the *default policy* (given a child how to simulate until a terminal node).

Algorithm 1 Monte Carlo Tree Search

```
1: function MctsSearch(s_0)
2:     create root node v_0 with state s_0
3:     while withing computational budget do
4:         v_l ← TreePolicy(v_0)
5:         △ ← DefaultPolicy(s(v_l))
6:         Backup(v_l, △)
7:     return a(BestChild(v_0))
```

Some of the most significant characteristics of MCTS over traditional tree search methods are:

- *Aheuristic*: MCTS does not require any domain-specific knowledge. That means we can achieve a good performance without using a heuristic function. This is because, we can run a simulation until reaching a terminal node, where it is enough with determining which player won the game. This is a huge advantage especially for complex games where useful heuristics are much more difficult to formulate. But the fact that we do not need a heuristic does not mean that we cannot take benefit from using one.
- *Anytime*: MCTS can stop at any time and return the most promising action until that moment. Like applying iterative deepening in *minimax* but with a much finer control.
- *Asymmetric*: The fact that we are selecting the most promising node of the tree with the *tree*

policy leads to an asymmetric tree over time. That is important to focus, and spend more time in the most relevant parts of the tree.

2.2 Strategy

This section reviews all AI techniques that have been used to address the problem of strategic decision making in RTS games, which are: hard-coded, planning-based or machine learning; and some techniques for strategy cooperation.

One of the most common techniques is using **hard-coded** approaches. These approaches usually encapsulate behaviors in states like "gather resources", "defend base" or "harass unit", and define conditions to trigger transitions between them. Although it is an effective way to create simple behaviors, AI authors have to foresee any possible game state to cover all the cases, something not always feasible. Some hard-coded approaches are basic Finite State Machines (FSM) [22], Hierarchical FSMs, or behavior trees (BT).

A technique that offers more adaptability than hard-coded approaches is **planning**. Unfortunately, planning approaches struggle with the real-time constrain of RTS games. Ontañón et al. [23] studied the possibility of using real-time Case-Based Planning (CBP) in the domain of Wargus (a WarCraft II clone) by learning plans from human demonstration, which are then composed at run-time in order to form full-fledged strategies to play the game. Mishra et al. [24] improved over their previous CBP approach by using situation assessment that improved the quality and speed of plan retrieval. Hierarchical Task-Network (HTN) planning has also been proposed for strategic decision making in RTS games [25]. Laagland [26] used a HTN to produce a non optimal build order, that it was able to beat the build-in AI of Spring RTS game (a "Total Annihilation" clone) due the better resource management. Sailer et al. [27] used a set of defined strategies and build the Nash equilibrium payoff matrix to choose the best strategy in interval times. Finally, Churchill and Buro [28] used planning in order to construct its economic build-orders, taking into account timing constraints of the different actions.

Many **machine learning** techniques have been explored. Weber and Mateas [29] used a supervised learning approach on labeled StarCraft replays to predict the player strategy. Hidden Markov Models (HMM) were used by Dereszynski et al. [30] to learn the transition probabilities of build order sequences in order to generate a probabilistic behavior models in StarCraft. Synnaeve and Bessière [31], using the dataset of Weber and Mateas [29], presented a Bayesian semi-supervised model to learn from replays and predict build orders from StarCraft replays; the replays were labeled by an EM clustering. They extended the previous work [32] presenting an unsupervised learning Bayesian model for tech-tree prediction from the same game replays.

A sub-category of machine-learning approaches is Case-Based Reasoning (CBR) [33]. CBR was used by Aha et al. [34] to perform dynamic plan retrieval in the Wargus domain. Inspired by the previous work, Hsieh and Sun [35] used StarCraft replays to construct states and build orders. Schadd et al. [36] used CBR and hierarchically structured models of the opponent behavior to perform opponent modeling. Jaidee et al. [37] studied the use of CBR for automatic goal selection to retrieve the proper Q-table in a reinforcement learning framework. Finally, Čertický et al. [38] used CBR to build armies, based on the opponent's army composition, and they pointed out the importance of proper scouting for better results.

In some RTS games you can form alliances to work as a team. We can categorize the **cooperation** in two groups: between AIs, or between an AI and a human. Magnusson and Balsasubramaniyan [39] created a StarCraft bot that cooperates with a human player. The bot inform about its intentions via chat and the player has options to order commands such us Attack, Follow, Drop or Expand. In their evaluation they found that players enjoyed having a collaborative AI and only experienced players liked the option to be able to give generic commands to their AI partner. Karlsson [40] studied the cooperation between AIs, showing in their experiments that a team of two collaborative AIs performs better than a team of two non collaborative AIs.

2.3 Tactics

Tactical reasoning usually involves a small group of units and how to command them to gain a tactical advantage. The most common approaches explored are: machine learning or game tree search.

Concerning **machine learning** approaches, Hladky and Bulitko [41] studied the use of Hidden Semi-Markov Models (HSMM) and particle filtering to track units over time. The experiments were performed in a First-Person Shooter (FSP) game, but the approach can be easily translated into RTS games. Kabanza et al. [42] improved the plan recognition system PHATT [43] by encoding strategies as HTNs; allowing them to predict tactical moves and devise counter actions. Sharma et al. [44] defined tactical plans as cases to be used in a CBR and reinforcement learning environment. Cadena and Garrido [45] proposed to use fuzzy techniques for tactical case matching in a CBR system. Synnaeve and Bessière [46] enhanced the regions generated by a map abstraction procedure (BWTA [47]) with tactical scores extracted from analyzing professional players replays. Finally, Miles [48] presented *IMTrees* for tactical decision making related to spatial reasoning; the idea is to use evolutionary algorithms to learn how to combine a set of basic influence maps.

In the category of **game tree search** approaches, Churchill and Buro [49] presented the ABCD algorithm (Alpha-Beta Considering Durations), a game tree search algorithm suitable for tactical combats in RTS games that is able to handle durative and simultaneous actions. Chung et al. [50] applied Monte-Carlo planning to a capture-the-flag version of Open RTS, and Balla and Fern [51] used a game state and action abstraction to reduce the complexity of the game tree in order to apply UCT (a Monte Carlo Tree Search algorithm) to tactical planning in Wargus. Chapter 3 describes our contribution and more detailed related work on adversarial game tree search.

2.4 Reactive Control

Reactive control is the way a player controls each individual unit to maximize its effectiveness in any type of situation. The response times required are usually very small since the game state is changing constantly.

Potential Fields and **Influence Maps** are similar techniques to achieve a reactive control behavior such as avoiding obstacles (navigation), avoiding opponent fire [52], or staying at maximum shooting distance [53]. A combination of A* and Potential Fields [54] was used to avoid local traps. Hagelbäck and Johansson [55] presented a bot that uses potential fields to navigate in games with fog-of-war like in the Tankbattle game. Avery et al. [56] and Smith et al. [57] had a similar idea to co-evolve influence map trees for spatial reasoning in RTS games. Danielsiek et al. [58] were able to coordinate squads to flank the enemy using influence maps in a RTS game. One of the drawbacks of potential field-based techniques is the large number of parameters to be tuned in order to achieve the desired behavior. Hence, we can find approaches to automatically learn these parameters such as reinforcement learning [59] or self-organizing-maps (SOM) [60]. However, notice that since potential fields are local search approaches, they are prone to make units stuck in local maxima.

One of the most popular approach for reactive control is to use some form of **machine learning**. Synnaeve and Bessière [61] presented how a Bayesian Model can be used to control units that it is able to receive tactical goals for directly. Some reinforcement learning (RL) [62] approaches have been explored. For instance, Wender and Watson [63] reviewed different RL algorithms for reactive control showing that all of them have similar performance. The main problem with RL approaches in the RTS domain is the large state space to explore, making it difficult to converge. Some of the proposed solutions are: (1) combine the Q-function of all the units in the group into a hierarchical Q-learning [64]; (2) use prior domain knowledge to cut down the search space [64, 65]; (3) learn just one Q-function for each unit type [66]; (4) using deep reinforcement learning to handle large state spaces (e.g., TorchCraft[1] [67]). For example, Usunier et al. [68] presented an algorithm for exploring deterministic policies in discrete action spaces using deep reinforcement learning; while Peng et al. [69] presented a bidirectionally-coordinated net (BiCNet) that combines policy networks

[1] `https://github.com/TorchCraft/TorchCraft`

and Q-networks; both works exhibit complex unit control and coordination. A "supervised" RL was proposed by Judah et al. [70] where a human critique was added in the learning loop that improves the learned policy. However, using only the human feedback the system achieves better results. Shantia et al. [71] used several multi-layer Neural Networks trained with SARSA [72] for target selection and retreat.

Evolutionary techniques have also been explored. Ponsen [73] tried to learn the parameters of the underlying model using an evolutionary learning techniques, but faced the same problem of dimensionality. Iuhasz [74] et al. defined a Neural Network for each unit that receives inputs like near units or obstacles and produce the action to attack or move. Then they use a neuroevolution algorithm called rtNEAT to evolve the NN topology and the connection weights. Finally, evolutionary optimization by simulating fights can easily be adapted to any parameter-dependent micro-management control model, as shown by [75] which optimizes an AIIDE 2010 micro-management competition bot.

CBR has been used again but in the context of reactive control. To achieve this, Szczepański and Aamodt [76] proposed several simplifications to the CBR process to speed up the retrieval: first, they only consider cases that meet some conditions, and secondly, they added reactionary cases that triggers as soon as certain conditions are meet. With this approach they were able to beat the built in AI of WarCraft III. More recently, Oh et al. [77] used influence maps as a similarity measure to retrieve combat cases, showing how their method was able to outperform most of the StarCraft competition bots.

Recently, Churchill et al. [49] explored the feasibility of using a **game tree search** approach (ABCD) for reactive control that was able to handle combat situations with up to eight versus eight units.

Pathfinding is a special case since it can be done in the tactical level, as the work presented by Danielsiek et al. [58], or in the reactive control level. The basic pathfinding algorithm A* is not enough to satisfy the complex, dynamic, real-time query with large number of units that RTS games exhibit. Therefore, new pathfinding algorithms keep appearing like: D*-Lite [78], hierarchical grids (HPA*) [79], Triangulation Reduction A* [80] or Jump Point Search [81]. For crowd navigation, rather than single units, we can find techniques such as *steering behaviors* [82] that can be used on top of a pathfinding algorithm; or *flow fields* [83] that have been used successfully in commercial RTS games like StarCraft 2 or Supreme Commander 2. A comprehensive review about (grid-based) pathfinding was recently done by Sturtevant [84].

2.5 Terrain Analysis

Terrain analysis supplies the AI with structured information about the map that can be exploited to make decisions. The analysis is usually performed offline in order to save CPU time during the game, but some online features can be extracted to capture dynamic changes. Pottinger [85] described the *BANG* engine implemented by Ensemble Studios for the game Age of Empires II. This engine provides terrain analysis functionalities to the game using influence maps and areas with connectivity information. Forbus et al. [86] showed the importance of analyzing game maps to have a qualitative spatial representation over which to perform reasoning. They used a Voronoi diagram to characterize the free space and use it for pathfinding. Hale et al. [87] presented a 2D geometric navigation mesh generation method from expanding convex regions from seeds. Perkins [47] developed a library called BWTA that using a Voronoi diagram as starting point, decomposes a map into regions and chokepoints. Even if BWTA is one of the most robust libraries, the decomposition process is slow (it takes more than one minute to analyze a map), and often fails to detect some chokepoints. Section 4.1 shows our improvements over the previous work into a new library (BWTA2). Higgins [88] used a different approach than Voronoi diagram and presented the idea of *auras* to expand regions and find chokepoints. Similarly, Halldórsson et al. [89] presented a *water level* approach (which it is discrete) to detect the chokepoints in order to use them as a gateways for the pathfinding heuristic. Although their approach is faster than Perkins, their algorithm tends to generate more false negative and false positive chokepoints. Bidakaew et al. [90] presented a map decomposition approach based on finding the medial axis and capable of detecting small corridors (what they call *area chokepoint*),

Table 2.1: Game tree properties in μRTS for two different map sizes.

	8x8	**12x12**
Leaf Correlation	0.994	0.987
Bias	0.510	0.511
Disambiguation Factor	-0.010	0.046

although this approach still detects some false negative/positive chokepoints. Finally, an on-line terrain analysis gathering terrain knowledge through exploration is also proposed by [91]

Walling is the act of intentionally placing buildings at the entrance of your base to block the path and to prevent the opponent's units from getting inside. This technique is used by human StarCraft players to survive early aggression and earn time to train more units. Čertický solved this constraint satisfaction problem using Answer Set Programming (ASP) [92]. Richoux et al. [93] improved the previous approach using a constrain optimization solver designed for real-time constrains (GHOST [94]). More details in Section 4.2. Finally, Oliveira and Madeira [95] used potential fields to guide the search to near optimal solutions.

2.6 Imperfect Information

Zermelo's theorem [96] says that an optimal **deterministic strategy** (a.k.a. a *pure strategy*) can be found computing the Nash equilibrium for perfect information games, but in imperfect information games deterministic strategies can be exploited by the opponent, rendering them suboptimal. This was confirmed by Kuhn [97] showing that the optimal strategy for a simplified poker game is indeed a **randomized strategy** (a.k.a. a *behavior strategy*).

Imperfect information games are usually modeled as **extensive-form games**, they are like perfect information games but with *information sets* to combine all the states (a.k.a. *worlds*) that are indistinguishable to a player at the time she has to make a decision. Unfortunately, the game tree complexity of imperfect information games tends to be very high even for simple games, and approaches to compute the optimal *randomized strategy* can be exponential on the size of the game tree [97] or at least polynomial in the size of the game tree [98]. For this reason most researchers tried approximation algorithms or they made assumptions related to their particular game that do not hold for RTS games. Some of them assume an opponent with full observability and they require to visit all the possible states, like *Best Defence model* [99], *Vector minimaxing* [100], or *Believe-state AND-OR tree search* [101]. Zinkevich et al. [102] presented *Counterfactual Regret Minimization* (CFR), an algorithm that converges to the Nash equilibrium and does not assume an opponent with full observability, but still needs to sample all the different states. Lanctot et al. [103] improved the previous algorithm to avoid sampling all the states using Monte Carlo sampling (*Monte Carlo CFR*, MCCFR). MCCFR has been applied with great success in games with a short game tree depth, compared to that in RTS games, and where the disambiguation of the information sets only happens at the end, such as Poker or Liar's Dice.

The most used simplification is known as *determinization* where the idea is to sample a *world* from the information set and proceed with a perfect information game tree algorithm. This is usually repeated several times, and the action to perform is determined via voting. This has been described as "averaging over clairvoyance" [104] and as Frank et al. [99] pointed out, it raises many problems: 1) players must behave the same way in states from the same information set (**strategy fusion**) and 2) the search can be "fooled" to pursue a highly rewarded state that cannot be reached under some information sets (**non-locality**). Additionally, in the case of imperfect information RTS games, *determinization* leads to **fake omniscience**, which happens when the player never tries to hide or gain information because she believes that she has perfect information. Despite these problems, *determinization* works very well in some domains. Long et al. [105] explained why by defining three properties of game trees that can lead to *strategy fusion* and *non-locality*: 1) probability of another terminal node with the same payoff value (**leaf correlation**), 2) probability that the game will favor

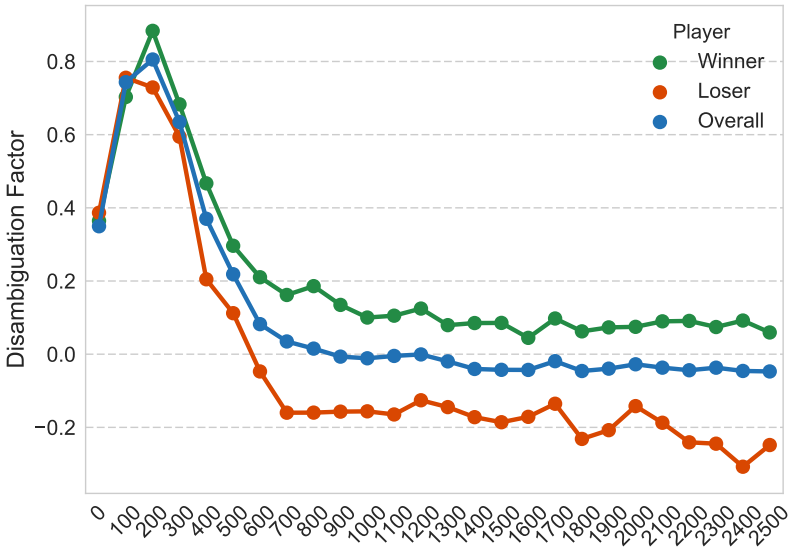


Figure 2.2: Disambiguation factor over time in μRTS using a 12x12 map.

a particular player over the other (**bias**), and 3) how quickly the states in an information set shrinks with regard to the depth of the tree (**disambiguation factor**). They found that *determinization* will perform well in games with a very high *disambiguation factor* or with a very high *leaf correlation* combined with a polarized *bias* (i.e., a very low or very high *bias*).

In order to see if determinization would work on RTS games, we estimated these values for μRTS. In RTS games, computing the exact values by exploring the whole game tree is not feasible, and thus, we estimated the values using an agent that makes random moves to partially explore the game tree in some small maps. Table 2.1 shows the approximated values after 10000 games in a 8×8 map and 2000 games in a 12×12 map, where we can do the following observations: 1) *leaf correlation* is very high as expected, since the last player's moves usually do not change the outcome of the game, 2) the *bias* is balanced, and 3) the *disambiguation factor*[2] is very low or even negative, this is because in RTS games we can lose information, hence the information sets can grow and not only shrink as stated in the definition. Figure 2.2 shows the average disambiguation factor for a winning player, for a losing player, and for any player, computed as the difference in the number of non-observable tiles (positive means that the number of non-observable tiles reduces, and negative that it grows). The figure shows how at the beginning of the game with a 12×12 map, both players start gaining information, but this gain decays over time. Also, we see that the loser player tends to lose information toward the end of a game (which is normal, since a losing player will start losing units fast at the end of the game). Therefore, according to [105], determinization would not work in RTS games; but we have to keep in mind that the *disambiguation factor* in RTS games is very low due the fact that we can lose information. Looking at Figure 2.2 we can observe how the *disambiguation factor* can reach very high values; which might be an indication that determinization could actually work in RTS games (our results in Section 3.6 actually show it does work).

As mentioned above, the most common technique to handle partial information in game tree search is *determinization*, here, we present a small overview of determinization algorithms:

- **Monte Carlo Sampling** [106]: Also known as *Perfect Information Monte Carlo Sampling* (PIMCS), it randomly samples states to apply a perfect information search algorithm like alpha-beta and it returns the best average move of all sampled states. This technique has been applied in games like Bridge [107], but it has been shown that the error to find the optimal

[2]Since calculating the size of the information set in a RTS game is complex, due to their size, we used the number of non-observable tiles in the map as a proxy; the assumption is that the size of the information set is exponential in the size of non-observable tiles.

strategy rapidly approaches 100% as the depth of the game tree increases (depth = 13) [108].

- **Statistical Sampling** [109]: Parker et al. proposed to use statistical sampling for large believe state games like Kriegspiel. More specifically they proposed 4 different sampling strategies: *Last Observation Sampling* (LOS), *All Observation Sampling* (AOS), *All Observation Sampling with Pool* (AOSP) and *Hybrid Sampling* (HS). For each sample state (world) it uses alpha-beta and decides the move that maximizes the score.

- **Information-set search** [110]: Parker et al. proposed to use game-tree search using *information sets* and computing the expected utility (EU) of each *information set*. The EU is computed as the weighted sum of the EUs for each possible move, weighted by the probabilities of a move given all the previous moves (perfect recall) times the EU of applying the move. To make the problem tractable in Kriegspiel, they used the following simplifications:

  - The states in each information set are sampled using Monte Carlo sampling.
  - The search depth bounded and a heuristic evaluation function is used for the EU.
  - The probabilities of choosing moves for the opponent are limited to two options: the opponent knows our pure strategy and choses moves that minimize our EU (**paranoia**); the opponent does not know anything and uses a uniform random distribution of our actions (**overconfidence**).

  In their experiments, the overconfident opponent model outperformed the paranoid model.

- **Monte Carlo Tree Search (MCTS) with Simulation Sampling** [111]: Ciancarini et al. proposed to delay the determinization until the simulation phase of MCTS, showing that they got better results using a heuristic function to get the probability of each type of world.

- **Determinized MCTS** [112]: It uses *root parallelization* where each root is a different determinization.

- **Single Observer Information Set MCTS (SO-ISMCTS)** [113]: The idea is to make a different random determinization on each iteration of MCTS to get the set of legal actions and then use information sets as nodes in the game tree. This algorithm makes several assumptions: 1) the same action (a) applied to all the states of an information set (I) transitions to the same information set (I'), 2) the opponent uses a random move selection on the moves that are not observed, and 3) at each real player turn (i.e., after executing an action in the game state), we can generate the information sets from the current observations. Unfortunately this last assumption is not true for RTS games because of the presence of durative actions.

- **Multi Observer Information Set MCTS (MO-ISMCTS)** [113]: To solve the second assumption of the previous algorithm, they proposed an alternative search method using two ISMCTS simultaneously, one for each player or "point of view". Traversing is done simultaneously but the action is considered from the point of view of each player. In their tests a *Determinized MCTS* works better for games with low probability of *strategy fusion* while MO-ISMCTS works better when there is a high chance of *strategy fusion*.

Although all of these techniques use *determinization* to some degree, those that compact tree nodes by information sets do not usually have the problem of *strategy fusion* or *non-locality*; and only SO-ISMCTS partially avoids *fake omniscience* since none of the players have perfect information but there is not a mechanism to detect and exploit gathering/hiding information actions. Another technique that does not use *determinization* is **Information-Theoretic Advisors** where Bud et al. [114] proposed to use three heuristic advisors to get the expected utility of a move: *strategic* advisor, *move hide* advisor and *seek hide* advisor. The *strategic* advisor returns the sum of the expected utility of the move in a specific world evaluated to a specific depth multiplied by the probability of the world for every possible world. The *move hide* advisor tries to maximize the opponent uncertainty (entropy) while *seek move* advisor tries to minimize our entropy. It assumes

the existence of an "Oracle" (they refer it as Game Controller and Distribution Maintenance Module) that knows the real game state and computes the probability distribution of the worlds for each player (and their entropy).

Little published research exists on partially observable RTS games. One instance of such work is that of Weber et al. [115], who designed a particle model to predict the locations of opponent's units under the fog-of-war in StarCraft. Another related line is that of strategies to perform scouting. Si et al. [91] assumed that the map is unknown beforehand and therefore the player must explore it. With this assumption they designed an on-line terrain analysis that gather terrain knowledge through exploration. In later work [116] they defined different scout strategies. Finally, Nguyen et al. [117] used potential flows to maximize the life time of the scouts to gather as much information as possible.

2.7 Holistic Approaches

There is a little work done trying to address the problem of building RTS AI using a single unified method, i.e., a holistic approach. The few exceptions are: (1) the Darmok system [118] using a combination of CBR and learning from demonstration; (2) a Darmok extension using behavior trees [119]; (3) ALisp [64]; or (4) Monte Carlo Tree Search based approaches that have been explored in the context of smaller-scale RTS games [2] or in a military-scale for full RTS games [120]. The main reason why there are few holistic approaches is because the complexity of RTS games is too large, and approaches that decompose the problem into smaller sub-problems have achieved better results in practice. Hence, techniques that scale up to large RTS games as StarCraft to control military and economic aspects are still not available.

A related problem is how to integrate the reasoning of different levels of abstraction. Molineaux et al. [121] showed how multi-scale goals and plans could be handled by case-based reasoning via an integrated RL/CBR algorithm using continuous models. While, Wander and Watson [63] proposed to solve it using a hierarchical CBR (HCBR) that connects different levels of abstraction like *Unit Navigation Level*, *Squad Level* and *Tactical Level*). A decompositional planning technique, similar to hierarchical task networks [122], called *Reactive Planning* [123] allows hierarchical plans that can be modified at different levels of abstraction. Synnaeve and Bessière [61] achieved a hierarchical goal structure by injecting the goal as another input in the sensor system.

Chapter 3: Adversarial Search

Performing adversarial search in RTS games requires addressing three problems: (1) the large search space as explained in Section 1.1.3; (2) the lack of a forward model; and (3) the fact that actions can be durative.

In order to reduce the **search space**, Section 3.1 proposes different methods to abstract the state space and the action space, showing that the selected abstraction can reduce the branching factor from a peak of 10^{410} to 10^{19} in the case of StarCraft.

Unlike classic board games like Chess where the output of each action is known, in commercial RTS games **forward models** are not available. In this thesis we limit ourselves to perform adversarial search to control only the military units, therefore our forward model only models combats. Section 3.2 reviews existing combat models (SparCraft, Lanchester's Laws, LTD2 and Bayes); and presents an extension of Lanchester's Square Law to model combats between heterogeneous armies (*TS-Lanchester*^{2}) and two new models based on DPF (*Sustained* and *Decreasing*). It also shows the feasibility of automatically learn the forward model from replay data in the case that unit's properties are unknown.

In Section 3.3 we experiment with algorithms that can handle **durative actions** (ABCD and MCTSCD) using our abstraction and forward model from the previous sections. Then, Section 3.4 presents an improvement over MCTSCD using an action Bayes model learned from replays to inform the two policies of MCTS (*tree policy* and *default policy*).

Finally, Section 3.5 presents an experimental evaluation of each of the previous contributions in this chapter; and Section 3.6 shows how to handle partial observability via a single believe state generation. Figure 3.1 illustrates what are the key components of the presented adversarial search, and how they match with the Game State-Player relation showed in Section 1.2.

3.1 Game State Abstraction

This section presents three different abstractions in order to reduce the state space and the action space of a RTS game:

- a **map abstraction** to represent the map as a graph where each node corresponds to a region of the map and edges represents adjacent regions;
- a **unit abstraction** to group all the units of the same type and in the same region into squads;
- and an **action abstraction** as a set of actions that a squad can perform over the abstract map.

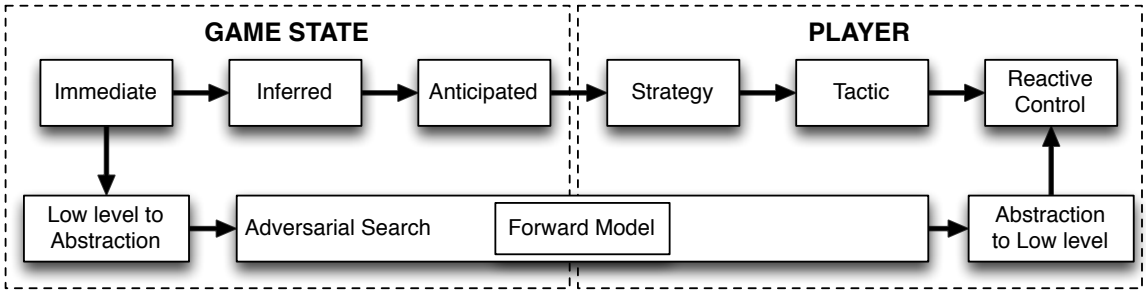
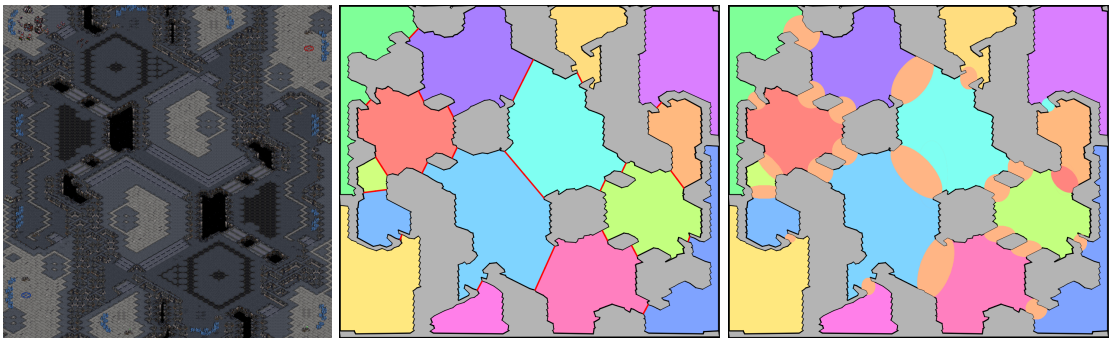


Figure 3.1: Mapping of the Game State Inference and Player Planning, to our Adversarial Search.



(a) StarCraft Benzene map. **(b)** Map abstraction with regions. **(c)** Map abstraction with regions and chokepoints.

Figure 3.2: Different map abstractions.

Finally, we present four variations of the abstraction and some experiments to evaluate the branching factor reduction of the abstracted game state over the original game state.

3.1.1 Map Abstraction

A RTS map is modeled as a graph where each node is a region R, and edges represent chokepoints (narrow passages) C between regions. Each region is associated with a specific set of walkable tiles in the original map of the game. As an example, Figure 3.2a shows the StarCraft map Benzene and Figure 3.2b the map abstracted in regions. The details of the map abstraction algorithm are presented in section 4.1 and implemented in the BWTA2 library. Since the map is invariant, this process has to be done only once, at the beginning of the game.

Most of RTS combats happen in the chokepoints locations, therefore we can conceivably expand the graph returned by turning each chokepoint into a region. This idea was also explored by Synnaeve and Bessière [124]. Chokepoint regions are added in the following way: given a set of regions R and a set of chokepoints C, a new region is created for each chokepoint $c \in C$ containing all the tiles in the map whose distance to the center of c is smaller than the radius of the chokepoint. Then, a pair of edges is added for each new chokepoint region to connect the original regions connected by the original chokepoint edge. An example of the result of this procedure is shown in Figure 3.2c.

Let us call **R** the map abstraction representing nodes as regions and **RC** the map abstraction representing nodes as regions and chokepoints. Table 3.1 shows, for a few StarCraft maps, the number of nodes in which each map is divided, the average connectivity of each node, and the diameter of the resulting graph.

3.1.2 Unit Abstraction

Instead of modeling each unit individually, units are grouped into **squads** by unit type and region. For each group, the following information is stored:

- *Player* is the owner of the squad,
- *Type* is the type of units in the squad,
- *Size* is the number of units in the squad,
- *Average HP* is the average hit points of all units in the squad,
- *Region*, which region is this squad in,
- *Action*, which action is this squad currently performing,
- *Target* is the target region of the action, if applicable and
- *End* is in which game frame is the action estimated to finish.

Table 3.2 shows an example of unit abstraction.

Table 3.1: Statistics of different StarCraft maps and map abstractions.

Map	Abs.	Size	Avg. Connect.	Diam.
Benzene	RC	59	1.492	8
Benzene	R	37	1.189	16
Destination	RC	65	1.538	8
Destination	R	40	1.250	16
Heartbreak Ridge	RC	68	1.471	9
Heartbreak Ridge	R	43	1.163	18
Aztec	RC	70	1.371	6
Aztec	R	46	1.043	12
Tau Cross	RC	58	1.379	6
Tau Cross	R	38	1.053	12
Python	RC	33	1.212	4
Python	R	23	0.870	8
Fortress	RC	29	1.103	4
Fortress	R	21	0.762	8
Empire of the Sun	RC	105	1.448	9
Empire of the Sun	R	67	1.134	18
Andromeda	RC	81	1.284	10
Andromeda	R	55	0.945	20
Circuit Breaker	RC	93	1.462	6
Circuit Breaker	R	59	1.153	12

3.1.3 Action Abstraction

Actions are abstracted to capture the actions that can be performed by a squad. The set of possible abstract actions are the following:

- *N/A*: only for buildings as they cannot perform any action,
- *Idle*: do nothing during 400 frames,
- *Attack*: attack enemies in the current region, and
- *Move*: move to an adjacent region. We distinguish between different types of *Move* using a set of boolean features depending on the properties of the target region to move:
  - **To Friend**: whether the target region has friendly units.
  - **To Enemy**: whether the target region has enemy units.
  - **Towards Friend**: whether the target region is closer to a friendly base than the current one.
  - **Towards Enemy**: whether the target region is closer to a enemy base than the current one.

  Notice that these features are not mutually exclusive, i.e., all 16 combinations are possible.

For example, the squads in Table 3.2 can execute the following actions:

- Bases: *N/A*.
- Tanks: Assuming that the Tanks were not executing any action, they could *Move* to region 1 or 3, or stay *Idle*. They cannot *Attack* since there is no enemy in region 2.
- Vultures: Assuming that the Vultures were not executing any action, they can *Move* to region 2, or stay *Idle*.

Since player 1 can issue any combination of those actions to her units, the branching factor of this example would be $(1) \times (2 + 1) \times (1 + 1) = 6$.

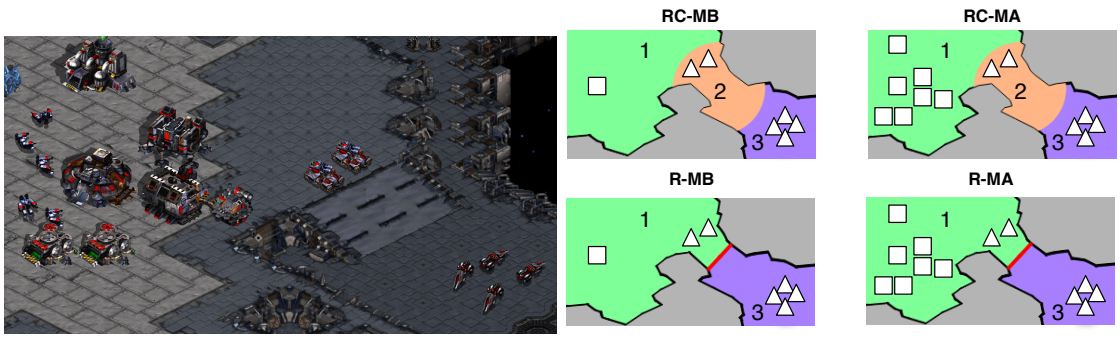


Figure 3.3: Snapshot of a StarCraft game and its representation with different high-level abstractions (numbers are the IDs of each region, triangles are military units, and squares are buildings).

Table 3.2: Grouped units of player 1 using abstraction RC-MB.

Plyr.	Type	Size	Avg. HP	Rgn.	Action	Target	End
1	Base	1	1500	1	N/A	-	-
1	Tank	2	150	2	Move	3	230
1	Vulture	4	80	3	Idle	-	400

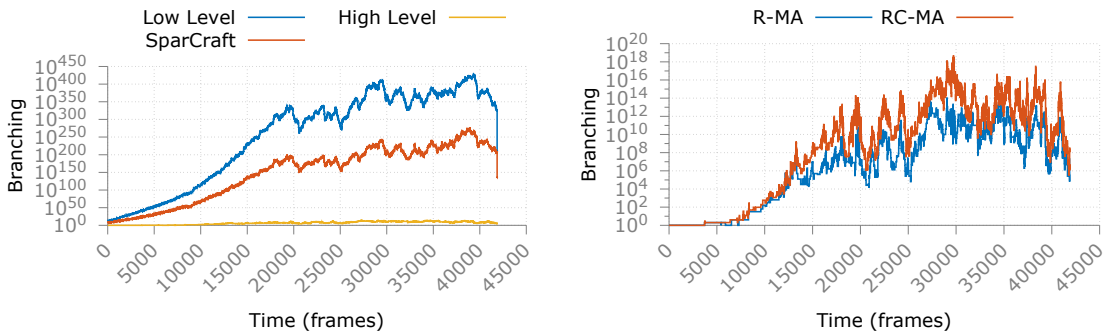
3.1.4 Experiments

Four abstract variations, combining the map abstraction and the unit abstraction, can be performed:

- **R-MB**: The map is abstracted into a graph of regions connected by chokepoints. All military units (except workers) are considered in the unit abstraction. The only structures considered are the bases (like *Command Centers* or *Nexus*).
- **R-MA**: Like R-MB, but including all structures.
- **RC-MB**: Like R-MB, but adding chokepoints as additional regions.
- **RC-MA**: Like RC-MB, but including all structures.

Figure 3.3 shows a screenshot of a situation in a StarCraft game, and graphically illustrates how this would be represented with the four different abstractions defined previously. The actual internal representation of the abstracted game state is a matrix with one row per unit group (Table 3.2). These four abstractions are used in section 3.5 to compare the performance of each one in the context of MCTS.

To compare the branching factor resulting from an abstract representation with that of the low-level representation, we analyzed the branching factor in StarCraft game states every frame during a game between professional players (Figure 3.4a): *Low Level* is the actual branching factor of StarCraft when all actions of all units are considered in the low-level game state representation, SparCraft is the branching factor of the low-level game state only considering combat units, and *High Level*, is the branching factor using the **RC-MA** abstraction. As expected, the branching factor using the low-level representation of the game state is very large (reaching values of up to 10^{410} in our experiment). This empirical evaluation reveals that in fact our approximation of a branching factor 10^{75} (Section 1.1.3) was conservative. On the other hand, our abstraction is able to keep the branching factor relatively low (reaching a peak of about 10^{19}). If we compare our proposed abstractions, there is not a significant difference between *-MA and *-MB, while adding the chokepoints as a new regions (R-* vs RC-*) increases the branching factor as we can see in Figure 3.4b.



(a) Comparison of the branching factor between low-level StarCraft, SparCraft and **R-MA** abstraction.

(b) Comparison of the branching factor between **R** and **RC** abstractions.

Figure 3.4: Branching factor of different game representations.

3.2 Forward Models for RTS Games

A significant number of AI algorithms assume the existence of a forward model for advancing (or predicting) the state after executing individual actions in the current game state. While this assumption is reasonable in certain domains, such as Chess or Go where simulating the effect of actions is trivial, precise descriptions of the effect of actions are not available in some domains. Thus, techniques for defining forward models are of key importance to RTS game AI, since they would allow the application of game tree search algorithms, such as MCTS.

This section focuses on the problem of defining "forward models" for RTS games, limiting ourselves to only simulate combat situations. We propose to model a combat as an *attrition game* [15] (an abstract combat game where individual units cannot move and only their damage and hit points are considered). We then propose three models (*TS-Lanchester*^{2}, *Sustained* and *Decreasing*) to simulate the evolution of an attrition game over time, and compare them with the different existing models in the literature. Finally, we identify the set of parameters that these models require, and study how to automatically learn those parameters from replay data. We argue that while forward models might not be available, logs of previous games might be available, from where the result of applying specific actions in certain situations can be observed. For example, consider the StarCraft game (that is used as our testbed), where precise definitions of the effects of unit actions are not available, but large collections of replays are.

We will use the term "combat model" to refer to a "forward model" in the context of a combat simulation. Moreover, notice that forward models can be used as evaluation functions (simulating the combat to the end, and then assessing the final state), and thus, we will also study the use of combat models as evaluation functions.

3.2.1 Existing Combat Models for StarCraft

Different types of combat models have been proposed in the literature for RTS games. In this section, we will first categorize those existing models into three broad classes: *low-level final state prediction models* (which try to model the combat as close as possible to the real game); *high-level final state prediction models* (which use some level of abstraction but try to give an accurate prediction of the remaining army composition at the end of a combat); and *high-level winner prediction models* (which only predict the winner).

Low-level Final State Prediction Models

A well known low-level model for StarCraft is SparCraft.[1] SparCraft was developed via a big human effort observing the behavior of different StarCraft units frame by frame. Despite being extremely accurate for small combat scenarios, SparCraft is not exhaustive (for example, it does not model unit collisions, and it does not support some unit types such as spell casters or transports) due to the tremendous amount of effort that it would take to model the complete StarCraft game. However, there has been recent work [125] on improving the accuracy of the unit movement simulation in SparCraft. Despite reaching a high degree of fidelity, a low-level model that predicts combat outcomes with 100% accuracy is not possible due some stochastic components of StarCraft. For example, each unit has a waiting time between attacks (a.k.a. "cooldown") that can be vary randomly, also, in some situations, units have only a chance of success when attacking a target. Thus, stochastic forward models would be required to accurately capture the underlying dynamics.

The biggest handicap of low-level models like SparCraft is computational cost. First, simulating a combat requires simulating all the individual details of the game state. Also, if these models are used as the representation for game tree search, the branching factor of the resulting search tree can be very high. The branching factor at the level of abstraction at which SparCraft operates can be as high as 10^{200} [126]. Another problem is that these models do not generalize well (i.e., SparCraft cannot be used to simulate WarCraft).

However, the advantages are that these models are able to simulate the effects of the actions precisely, and the input for the model is the current game state without any preprocessing.

High-level Final State Prediction Models

In order to reduce computational cost, combat models over game state abstractions have been proposed. For example, Soemers [127] proposed a combat model based on Lanchester's Square Law. Lanchester's Laws [128] assume that combats are between two homogeneous armies; a *combat effectiveness* (or Lanchester attrition-rate coefficient) for each army is computable; and any unit in an army can attack any opponent unit. However, this is not the case in most RTS games where, for example, not all units can attack flying units and armies can be heterogeneous. To overcome these shortcomings, several extensions have been proposed [129], but only some of them have been explored in the context of RTS games. Soemers computed the combat effectiveness of heterogeneous armies aggregating the mean DPF (Damage per Frame) of an army, divided by the mean HP of the opponent's army. Similarly, Stanescu et al. [130] used the Lanchester's Generalized Law (a generalization of the Square Law), but this time learning the combat effectiveness from replay data. As a consequence of only considering homogeneous armies, Lanchester's Laws can predict the size of the surviving army at the end of a combat, but not which specific units survived (since armies are assumed to be homogeneous). Stanescu et al. only used this model to predict the winner of a combat, and thus did not address this problem. Soemers addressed this by determining which units survived the combat in a heterogeneous army by selecting the remaining units randomly (while matching with the predicted army size).

The advantages of high-level state prediction models are that they offer a faster simulation time than low-level models, at the price of losing some low-level details due to abstraction.

High-level Winner Prediction Models

Finally, some approaches focus directly on just predicting the winner of a combat, instead of providing a prediction of the final state of the combat. These models can be classified in two subcategories: hand-crafted models and machine learned models.

Concerning *hand-crafted models*, Uriarte [131] defined a basic heuristic assuming that both armies continuously deal their starting amount of DPF to each other, until one of the armies is destroyed. Kovarsky and Buro [132] proposed a function that gives more importance to having multiple units

[1] `https://github.com/davechurchill/ualbertabot/tree/master/SparCraft`

with less HP than only one unit with full HP: Life Time Damage 2 (LTD2).

$$LTD2 = \sum_{u \in A} \sqrt{u.HP} \times u.DPF - \sum_{u \in B} \sqrt{u.HP} \times u.DPF$$

Nguyen et al. [133] considered that RTS games properties are non-additive, i.e., if $\mu(X)$ is the effectiveness by a unit combinations X, $\mu(X_1 \cup X_2) > \mu(X_1) + \mu(X_2)$ or $\mu(X_1 \cup X_2) < \mu(X_1) + \mu(X_2)$; and they proposed to use fuzzy integrals to model this aggregation problem.

Concerning *machine learned models*, Synnaeve and Bessière [134] clustered armies based on their unit compositions and they were able to predict the winner using a Gaussian mixture model; Stanescu et al. [135] improved the previous work defining a Bayesian network and using a Gaussian Density Filtering to learn some features of the model. Finally, Sánchez-Ruiz [136] experimented with some machine learning algorithms (such as LDA, QDA, SVM or kNN) to predict the winner of small combats over time and in a more recent work [137] Sánchez-Ruiz et al. used influence maps to cluster each player and make winner predictions of games with more than two players.

The key advantage of these models is their low computational cost, while being able to be used as *evaluation functions*; and the option to train them.

3.2.2 Proposed Combat Models for StarCraft

One of the mechanics present in almost all RTS games is combat. In a combat situation each player commands an army of units to defeat the opponent. This section presents three new high-level final state prediction models, based on attrition games. The aim of the proposed models is to predict the final state of a combat with higher accuracy and/or better performance than existing models. Additionally, in order to be useful in the context of game tree search, these models can: 1) predict the final state of a combat (which units survived, even for heterogeneous army compositions), 2) predict the duration of a combat, and 3) simulate partial combats (i.e., not only fights-to-the-end).

Combat Models as Attrition Games

We propose to model a combat as an *attrition game* [15]. An *attrition game* is a two-player simultaneous-move game played on a directed graph, in which each node corresponds to a unit health, attack power and it belongs to a player. A directed edge (x, y) denotes that node x can deal damage to node y. When the health of a node reaches to zero the node is removed from the graph. The goal of the game is to destroy all opposing nodes. At each round each node can select a target and at the end of the round all the damage is done simultaneously. Furtak and Buro proved that finding the optimal strategy of an *attrition game* is **PSPACE**-hard [15]. This abstraction can be seen as a target selection problem since no movement is involved in the process and nodes do not have complex properties. A similar combat abstraction was studied by Kovarsky and Buro [132] in the context of heuristic search.

Using *attrition games* as a starting point, we define a combat as a tuple $C = \langle A, B, E \rangle$, where:

- A and B are the sets of *units* (or nodes) of each player,
- a *unit* is a tuple $u = \langle h, ap \rangle$, where h is the health of the unit and ap its attack power,
- E is the set of directed edges between units (there is an edge from a unit a to a unit b, when a can attack b).

In the particular case of StarCraft, a unit is extended to be a tuple $u = \langle type, pos, hp, s, e \rangle$, where:

- *type* is the unit type, which defines some common properties like maximum Hit Points (HP), weapon damage, or weapon "cooldown" (time between attacks),
- $pos = \langle x, y \rangle$ is the unit position (used to determine which units are inside the attack range of other units),
- hp are the hit points, s is the shield energy, and e is the spell energy.

In our model, the node's *attack power* captures both the damage dealt by each attack, as well as the cooldown time between attacks, and is determined by the *unit type* (to get the weapon damage and cooldown, or the *energy* to know if the unit is ready to cast a spell). To define the node's health we use the *unit type* to know the shield value, and the current *hit points* and *shield* energy. Finally, a *combat model* m is a function that given an initial combat state at time 0 ($\langle A_0, B_0, E\rangle$) predicts the length of the combat (t) and the final combat state (A_t, B_t): $m : \langle A_0, B_0, E\rangle \rightarrow \langle A_f, B_f, t\rangle$.

Target-Selection Lanchester's Square Law Model (*TS-Lanchester*^{2})

TS-Lanchester^2 is related to the model proposed by Soemers [127] in that they are both based on Lanchester's Square Law [128], but differs in two key aspects: First, *TS-Lanchester*^{2} is based on the original formulation of Lanchester's Square Law (as formulated in [129]), while Soemers uses a reformulation in order to support army reinforcements. Second, and most important, our model incorporates a target selection policy, used at the end of the simulation to determine exactly which units remain at the end of the combat.

Since we use the original formulation of the Square Law instead of the formulation used by Soemers, we include the exact version of the equations used by our model here. Specifically, we model a combat as:

$$\frac{\mathrm{d}|A_t|}{\mathrm{d}t} = -\alpha|B_t|, \frac{\mathrm{d}|B_t|}{\mathrm{d}t} = -\beta|A_t|$$

where $|A_t|$ denotes the number of units of army A at time t, $|B_t|$ is the analogous for army B, α is the rate at which a unit in B can kill a unit in A (a.k.a *combat effectiveness or Lanchester attrition-rate coefficient*) and β is the analogous for army A. The intensity of the combat is defined as $I = \sqrt{\alpha\beta}$.

Given an initial combat state $\langle A_0, B_0\rangle$, that is the list of player A units' and player B units', the model proceeds as follows:

1. First the model computes the *combat effectiveness* of an heterogeneous army for each army as:

$$\alpha = \frac{\overline{DPF}(B, A)}{\overline{HP}(A)}, \beta = \frac{\overline{DPF}(A, B)}{\overline{HP}(B)}$$

   where $\overline{HP}(A)$ is the average HP of units in army A and $\overline{DPF}(B, A)$ is the expected DPF that units in army B can deal to units in army A computed as follows:

$$\overline{DPF}(B, A) = \frac{\overline{DPF}_a(B)HP_a(A) + \overline{DPF}_g(B)HP_g(A)}{HP_a(A) + HP_g(A)}$$

   where $HP_a(A)$ and $HP_g(A)$ is the sum of HP of the air and ground units of an army respectively, and $\overline{DPF}_a(B)$ and $\overline{DPF}_g(B)$ is the average DPF of the air and ground units of army B respectively.

2. The winner is then predicted:

$$\frac{|A_0|}{|B_0|} \begin{cases} > R_\alpha & A \text{ wins} \\ = R_\alpha & \text{draw} \\ < R_\alpha & B \text{ wins} \end{cases}$$

   where $|A_0|$ and $|B_0|$ are the number of initial units in army A and B respectively, and R_α is the relative effectiveness $\left(R_\alpha = \sqrt{\frac{\alpha}{\beta}}, R_\beta = \sqrt{\frac{\beta}{\alpha}}\right)$.

3. The length of the combat is computed as:

$$t = \begin{cases} \frac{1}{2I} ln \left(\frac{1 + \frac{|B_0|}{|A_0|} R_\alpha}{1 - \frac{|B_0|}{|A_0|} R_\alpha} \right) & \text{if } A \text{ wins} \\ \frac{1}{2I} ln \left(\frac{1 + \frac{|A_0|}{|B_0|} R_\beta}{1 - \frac{|A_0|}{|B_0|} R_\beta} \right) & \text{if } B \text{ wins} \end{cases}$$

   Notice, however, that since this is a continuous model, the special case where there is a tie (both armies annihilate each other) is problematic, and results in a predicted time $t = \infty$. In our implementation, we detect this special case, and use our *Sustained* model (Section 3.2.2) to make a time prediction instead.

4. Once the length is known, the remaining units are:

$$\begin{cases} |A| = \sqrt{|A_0|^2 - \frac{\alpha}{\beta}|B_0|^2} & |B| = 0 \quad \text{if } A \text{ wins} \\ |B| = \sqrt{|B_0|^2 - \frac{\beta}{\alpha}|A_0|^2} & |A| = 0 \quad \text{if } B \text{ wins} \end{cases}$$

   To determine which units survived, the model uses a *target selection policy*, which determines in what order the units are removed. The final combat state $\langle A_f, B_f, t \rangle$ is defined as the units that were not destroyed at time t.

This model predicts the final state given a time t as follows:

$$|A_t| = \frac{1}{2} \left((|A_0| - R_\alpha |B_0|) \, e^{It} + (|A_0| + R_\alpha |B_0|) \, e^{-It} \right)$$

$$|B_t| = \frac{1}{2} \left((|B_0| - R_\beta |A_0|) \, e^{It} + (|B_0| + R_\beta |A_0|) \, e^{-It} \right)$$

TS-Lanchester^2 has two input parameters: a vector of length k (where k is the number of different unit types, in StarCraft $k = 163$) with the DPF of each unit type; and a *target selection policy*. In Section 3.2.3, we propose different ways in which these input parameters can be generated.

Sustained DPF Model (*Sustained*)

Sustained is an extension of the model presented by Uriarte in [131]. It assumes that the amount of damage an army can deal does not decrease over time during the combat (this is obviously an oversimplification, since armies might lose units during a combat, thus decreasing their damage dealing capability) but it models which units can attack each other with a greater level of detail than *TS-Lanchester*^{2}.

Given an initial combat state $\langle A_0, B_0 \rangle$, where armies belong to each player, the model proceeds as follows:

1. First, the model computes how much time each army needs to destroy the other. In some RTS games, such as StarCraft, units might have a different DPF (*damage per frame*) when attacking to different types of units (e.g., air vs ground units), and some units might not even be able to attack certain other units (e.g., walking swordsmen cannot attack a flying dragon). Thus, for a given army, we can compute its DPF_{air} (the aggregated DPF of units with a weapon that can attack air units), DPF_{ground} (the aggregated DPF of units with a ground weapon) and DPF_{both} (aggregated DPF of the units with both weapons, ground and air). After that, the model computes the time required to destroy all air and ground units separately:

$$t_{air}(A, B) = \frac{HP_{air}(A)}{DPF_{air}(B)}$$

$$t_{ground}(A,B) = \frac{HP_{ground}(A)}{DPF_{ground}(B)}$$

where $HP(A)$ is the sum of the hit points of all the units. Then, the model computes which type of units (air or ground) would take longer to destroy, and DPF_{both} is assigned to that type. For instance, if the air units take more time to kill (i.e., $t_{air}(A,B) > t_{ground}(A,B)$) we recalculate $t_{air}(A,B)$ as:

$$t_{air}(A,B) = \frac{HP_{air}(A)}{DPF_{air}(B) + DPF_{both}(B)}$$

These equations are symmetric, therefore $t_{air}(B,A)$ is calculated analogously to $t_{air}(A,B)$. Finally, the global time to kill the other army is computed:

$$t(A,B) = max(t_{air}(A,B), t_{ground}(A,B))$$

2. The combat length time t is computed as:

$$t = min(t(A_0,B_0), t(B_0,A_0))$$

3. After that, the model computes which units do each army have time to destroy in time t. For this purpose, this model takes as input a *target selection policy*, which determines the order in which an army will attack the units in the groups of the other army. The final combat state $\langle A_f, B_f, t\rangle$ is defined as all the units that were not destroyed at time t.

Sustained has two input parameters: a vector of length k with the DPF of each unit type; and a *target selection policy*. Moreover, compared to *TS-Lanchester*^{2}, *Sustained* takes into account that not all units can attack all other units (e.g., ground versus air units). On the other hand, it does not take into account that as the combat progresses, units die, and thus the DPF of each army decreases, which is precisely what the model presented in the following subsection aims to address.

Decreasing DPF Model (*Decreasing*)

Decreasing is more fine grained than *Sustained*, and considers that when a unit is destroyed, the DPF that an army can deal is reduced. Thus, instead of computing how much time it will take to destroy the other army, it only computes how much time it will take to kill one unit, selected by the *target selection policy*. The process is detailed in Algorithm 2, where first the model determines which is the next unit that each player will attack using the *target selection policy* (lines 2-5); after that it computes the expected time to kill the selected unit using TIMETOKILLUNIT(u, A, DPF) (where u is the unit that army A will try to kill, and DPF is a matrix specifying the DPF that each unit type can deal to each other unit type); the target that should be killed first is eliminated from the combat state, and the HP of the survivors is updated (lines 16-25). The model keeps doing this until one army is completely annihilated or it cannot kill more units.

Decreasing has two input parameters: a $k \times k$ matrix DPF (where k is the number of different unit types), where $DPF_{i,j}$ is the DPF that a unit of type i would deal to a unit of type j; and a *target selection policy*. Let us now focus on how these parameters can be acquired for a given RTS game.

3.2.3 Combat Models Parameters

All three proposed models take as input the unit's DPF (Damage per Frame) and a *target selection policy*. This sections shows how these parameters can be calculated. For each of the two parameters, we performed experiments by learning these parameters from replay data (assuming no information about the game rules), and also using a collection of baseline configurations for them (e.g., taking the DPF values directly from the StarCraft manual).

Algorithm 2 Decreasing.

```
1:  function DECREASING(A, B, DPF, targetSelection)
2:      SORT(A, targetSelection)
3:      SORT(B, targetSelection)
4:      i ← 0                                              ▷ index for army A
5:      j ← 0                                              ▷ index for army B
6:      while true do
7:          t_b ← TIMETOKILLUNIT(B[j], A, DPF)
8:          t_a ← TIMETOKILLUNIT(A[i], B, DPF)
9:          while t_b = ∞ and j < B.size do
10:             j ← j + 1
11:             t_b ← TIMETOKILLUNIT(B[j], A, DPF)
12:         while t_a = ∞ and i < A.size do
13:             i ← i + 1
14:             t_a ← TIMETOKILLUNIT(A[i], B, DPF)
15:         if t_b = ∞ and t_f = ∞ then break
16:         if t_b = t_a then                              ▷ draw
17:             A.ERASE(i)
18:             B.ERASE(j)
19:         else
20:             if t_b < t_a then                          ▷ A wins
21:                 A[i].HP ← A[i].HP − DPF(B) × t_b
22:                 B.ERASE(j)
23:             else                                       ▷ B wins
24:                 B[j].HP ← B[j].HP − DPF(F) × t_a
25:                 A.ERASE(i)
26:         if i >= A.size or j >= B.size then break
27:     return A, B
```

- **Effective DPF**. In practice, the DPF a unit can deal does not just depend on the damage and cool-down of their weapon, but is also affected by their maneuverability, special abilities, the specific enemy being targeted, and the skill of the player. Therefore, we call *effective DPF* to the real DPF that a unit can actually deal in a real game situation. We will compute effective DPF at two different levels of granularity:
  - Per unit: a vector of length k (the number of unit types) with the effective DPF of each unit type.
  - Per match-up: a $k \times k$ matrix DPF, where $DPF^m(i,j)$ represents the DPF that a unit of type i is expected to deal to a unit of type j.

  Moreover, the $DPF^u(i)$ vector can be derived from the $DPF^m(i,j)$ matrix as follows:

  $$DPF^u(i) = \min_j DPF^m(i,j)$$

  Here, we use the *min* DPF rather than the *average* DPF, since that worked best in our experiments. Below we will focus on just how the effective DPF matrix can be learned from game replays. For comparison purposes, we will compare this against the **static** DPF ($u.damage/u.cooldown$) calculated directly from the StarCraft unit stats.

- **Target selection policy**. This is a function that given a set of units A, determines which unit $u \in A$ to attack next. We will also show how this can be learned from replays, and will compare the learned policy against two baselines: **random**, which selects units randomly from

a uniform distribution; and maximizing the **destroy score**. In StarCraft the unit's destroy score is approximately $2 \times u.mineralCost + 4 \times u.gasCost$ (although some units a higher or lower score, based on their special abilities).

Learning Combat Parameters

This section presents how to apply an off-line learning method from game replays to learn the combat parameters. Given a collection of replays, they can be preprocessed in order to generate a training set consisting of all the *combats* that occur in these replays. Given a training set consisting of a set of combats, the combat model parameters can be learned as follows:

- **Effective DPF** matrix. For each combat in the dataset, the following steps are performed:
  1. First, for each player p, count the number of units that can attack ground units (n^p_{ground}), air units (n^p_{air}) or both (n^p_{both}).
  2. Then, let K be a set containing a record (t_i, u_i) for each unit destroyed during the combat, where t_i is the frame where unit u_i was destroyed. For each $(t_i, u_i) \in K$, the total damage that had to be dealt up to t_i to destroy u_i is: $d_i = u_i.HP + u_i.shield$ where *HP* and *shield* are the hit points and shield of the unit at the start of the combat. We estimate how much of this damage was dealt by each of the enemy units, by distributing it uniformly among all the enemies that could have attacked u_i. For instance, if u_i is an air unit, the damage is split as:
  $$d_i^{split} = \frac{d_i}{n^p_{air} + n^p_{both}}$$
  After that, for each unit u that could attack u_i, two global counters are updated:
  $$damageToType(u, u_i) += d_i^{split}$$
  $$timeAttackingType(u, u_i) += t_{i-1} - t_i$$
  where t_{i-1} is the time at which the previous unit of player p was destroyed (or 0, if u_i was the first).
  3. After all the combats in the dataset have been processed, the effective DPF matrix is estimated as:
  $$DPF(i, j) = \frac{damageToType(i, j)}{timeAttackingType(i, j)}$$
- **Target selection policy**. We used a **Borda Count** method [138] to estimate the target selection policy used by the players in our dataset. The idea is to iterate over all the combats in the training set and each time a type of unit is killed for the first time, give that type $n - i$ points where n is the number of different unit types in the army and i the order the units were killed. For example, if an army is fighting against marines, tanks and workers ($n = 3$) and if the first units to be destroyed are tanks, then marines and then workers, the assigned scores will be: 2 points for the tank, 1 point for the marines and 0 points for the workers. After analyzing all the combats the average Borda Count of each unit type is computed and this is the score to sort the targets in order of preference.

  We repeated this estimation three times (generating three different target selection policies): one for the case when the attacking army is only composed by ground units, another one for armies with only air units and the last one for mixed armies. We observed, as expected, that the attacking armies composed by only ground units, prioritize enemies with ground weapons; and vice-versa for armies with only air units.

3.2.4 Extracting Combats from Game Replays

To be able to learn the parameters required by the combat models from replays, we need a dataset of combats. Other researchers achieved this by creating artificial datasets making the default

StarCraft AI to play against itself [136] and recording the state in traces; or using a crafted combat model (SparCraft) to run thousands of combat situations [130, 135], while others used data from professional human players [29, 134, 139]. In this dissertation we use the latter, to capture the performance of each unit as commanded by an expert.

Combat Records

Let us define a *combat record* as a tuple $CR = \langle t_0, t_f, R, A_0, B_0, A_f, B_f, K, P \rangle$, where:

- t_0 is the initial frame when the combat started and t_f the final frame when it finished,
- R is the reason why the combat finished. The options are:
  - **Army destroyed**. One of the two armies was totally destroyed during the combat.
  - **Peace**. None of the units were attacking for the last x frames. In our experiments $x = 144$, which is 6 seconds of game play.
  - **Reinforcement**. New units participating in the battle. This happens when units, that were far from the battle when it started, begin to participate in the combat.
  - **Game end**. This occurs when a combat is still going on, but one of the two players surrenders the game.
- A_0 and B_0 are the armies of each player at the start of the combat, and A_f and B_f are the armies of each player at the end of the combat.
- $K = \{(t_1, u_1), \ldots, (t_n, u_n)\}$ where t_i is the time when unit u_i was killed.
- P is the list of passive unites, i.e., units that did not participate in the combat, like workers keep mining while being attacked.

Given this definition, let us now introduce our approach to identify combats in StarCraft replays.

Detecting Combat Start

One of the hardest part is how to define *when* a combat starts and ends. In Synnaeve and Bessière's work [134] they consider a new combat when a unit is killed. Although this helps to prune all non combat activities involved in a RTS game, in some cases we are losing useful information. For example, imagine the situation when a ranged unit is kiting a melee unit and the melee unit is killed at the end. If we start tracking the combat after the kill, it will look like the ranged unit killed the melee unit without effort, when in fact it took a while and a smart move-and-hit behavior. Hence, we start tracking a new combat if a *military unit* is *aggressive* or *exposed* and not already in a combat:

- A *military unit* is a unit that can deal damage (by a weapon or a magic spell), detect cloaked units, or a transporter.
- A unit is *aggressive* when it has the order to attack or is inside a transport.
- A unit is *exposed* if it is in the attack range of an *aggressive* enemy unit.

Units Involved in Combat

Any military unit u that is either aggressive or exposed and not already in a combat, will trigger a new combat. To know which units are part of the combat triggered by u, let us define $inRange(u)$ to be the set of units in attack range of a unit u. Now, let $D = \cup_{u' \in inRange(u)} inRange(u')$. A_i is the subset of units of D belonging to player A at time t_i; and B_i is the subset of units of D belonging to player B at time t_i. Figure 3.5 shows a representation of a combat triggered by unit u (the black filled square) and the units in the combat (the filled squares). Notice that a *military unit* is considered to take part in a combat even if at the end it does not participate, but it will be marked as *passive*.

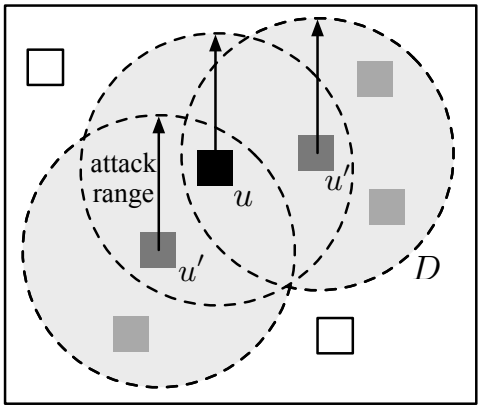


Figure 3.5: Black filled square (u) triggers a new combat. Only filled squares are added to the combat tracking, i.e., the union of all $inRange(u')$.

Combat Dataset

Now that we have a definition of a *combat record*, a dataset can be generated directly by processing replays. StarCraft replays of professional player games are widely available, and several authors have compiled collections of such replays in previous work [29, 134]. Since StarCraft replays only contain the mouse click events of the players (this is the minimum amount of information needed to reproduce the game in StarCraft), we do not know the full game state at a given point (no information about the location of the units or their health). Thus, we need to run the replay in StarCraft and record the required information using BWAPI.[2] This is not a trivial task since if we record the game state at each frame, we might have too much information (some consecutive recorded game state variables might have the same value) and the size of the data captured can grow too much to be processed efficiently. Some researchers proposed to capture different information at different resolutions to have a good trade-off of information resolution. For example, Synnaeve and Bessière [134] proposed recording information at three different levels of abstraction:

- General data: all BWAPI events (like unit creation, destruction or discovery). Economical situation every 25 frames and attack information every frame. It uses the heuristic explained above to detect attacks.
- Order data: all orders to units. It is the same information we will get parsing the replay file outside BWAPI.
- Location data: the position of each unit every 100 frames.

On the other hand, Robertson and Watson [140] proposed a uniformed information gathering, recording all the information every 24 frames or every 7 frames during attacks to have a better resolution than the previous work.

In our case we only need the combat information, so we used the analyzer from Synnaeve and Bessière, but with our proposed heuristic for detecting combats (described above). [3]

3.2.5 Experimental Evaluation

In order to evaluate the performance of each combat model we first collected a dataset of combats from game replays; we then compared the winner prediction accuracy over the combat dataset; after that we evaluated the accuracy of predicting the final state.

[2] `https://github.com/bwapi/bwapi`

[3] Source code of our replay analyzer, along with our dataset extracted from replays, can be found at `https://bitbucket.org/auriarte/bwrepdump`

Combat Dataset

We extracted the combats from 600 replays played by professional human players (100 for each race matchup: Terran versus Terran, Terran versus Protos, etc.). This resulted in a dataset of 140,797 combats where:

- 99,598 combats ended by *reinforcement*,
- 40,820 combats ended in *peace*,
- 11,454 combats ended with one *army destroyed*,
- 653 combats ended by *game end*.

To evaluate the performance of the combat models, we only used the subset of combats that ended with one *army destroyed*. To form the training set, we also removed combats with Vulture's mines (to avoid problems with friendly damage) and combats where an army was passive (none of their units fought back). This resulted in a dataset of 6,066 combats with an average combat length of 169.68 frames (max 3,176, min 1, combats with tiny durations are because of 1-shot-kills), an average of 9.4 units per combat (max 93, min 2) and an average of 2.95 different types of units per combat (max 10, min 2). Unfortunately, SparCraft does not support all types of units, therefore we also generated a subset of 4,558 combats that are valid for SparCraft in order to compare results against SparCraft. When using learned parameters (DPS learn or Borda Count target selection), we report the result of a 10-fold cross validation.

Winner Prediction Accuracy

In this first experiment we assessed the winner prediction accuracy of each combat model. When comparing against SparCraft, we simulate the combat in SparCraft by using one of the predefined scripted behaviors (*Attack-Closest (AC)*, *Attack-Weakest (AW)*, *Attack-Value (AV)*, *Kiter-Closest (KC)*, *Kiter-Value (KV)*, and *No-OverKill-Attack-Value (NOK-AV)*). For our proposed models (*TS-Lanchester*^{2}, *Sustained* and *Decreasing*) we experimented with two DPF matrices: *static* and *learn* as described in Section 3.2.3. When predicting the winner, the target selection policy only affects the *Decreasing* model; therefore we experimented with all three target selection policies (Random, Destroy Score and Borda Count) only for the *Decreasing* model. Finally, we also compare against evaluating the initial combat state with the LTD and LTD2 evaluation functions, and predicting the winner based on the resulting evaluation. Since LTD and LTD2 use the unit's DPF, we tested the models with both DPF matrices (static and learn). Notice that LTD and LTD2 use the actual unit's HP from the low-level game state (like SparCraft), in contrast of *TS-Lanchester*^{2}, *Sustained* and *Decreasing* use the average HP of each group of units.

Table 3.3 shows the winner prediction accuracy and the standard error of the mean (SEM) of each combat model in the full dataset (6,066 combats) and in the SparCraft compatible dataset (4,558 combats). A Z-test reveals that there is no statistical significant difference between *TS-Lanchester*^{2} and *Sustained* in the full dataset ($z = 1.146$); and LTD2, *TS-Lanchester*^{2}, and *Sustained* perform similarly for the SparCraft dataset (with z values lower than 1.764). Static DPF tends to result in higher performance than the learned DPF (although prediction accuracy is still very high with the learned DPF, showing it is a viable alternative for domains for which DPF of units is not known in advance). Concerning target selection policies, since this is a winner prediction task, they only play a role on the *Decreasing* model, and we see that except for one anomaly (*Decreasing* static with Borda count), they all perform similarly. Overall, the configuration that achieved best results is *TS-Lanchester*^{2} static.

The main conclusion that can be drawn is that models that learn parameters from data can get a level of performance of the static models (which assume all the domain parameters are known). Concerning the apparent low performance of SparCraft in our experiments, notice that Table 3.3 shows performance in predicting human vs human combats, whose behavior might differ significantly from the set of scripts used in SparCraft. SparCraft performance might differ if using a dataset representing bot behavior.

Table 3.3: Winner prediction accuracy and standard error (SEM) of each combat model

Combat Model	Full Dataset	SparCraft Dataset
SparCraft (AC)	N/A	88.02% ± 0.48
SparCraft (AW)	N/A	87.76% ± 0.49
SparCraft (AV)	N/A	87.76% ± 0.49
SparCraft (NOK-AV)	N/A	88.02% ± 0.48
SparCraft (KC)	N/A	84.69% ± 0.53
SparCraft (KV)	N/A	84.82% ± 0.53
LTD static	90.37% ± 0.38	91.82% ± 0.41
LTD learn	82.06% ± 0.16	84.20% ± 0.18
LTD2 static	92.09% ± 0.35	**93.22% ± 0.37**
LTD2 learn	85.30% ± 0.32	86.84% ± 0.29
TS-Lanchester^2 static	**94.10% ± 0.30**	**94.45% ± 0.34**
TS-Lanchester^2 learn	90.69% ± 0.12	91.29% ± 0.14
Sustained static	**93.60% ± 0.31**	**94.12% ± 0.35**
Sustained learn	90.40% ± 0.13	91.01% ± 0.14
Target Selection Policy: Random		
Decreasing static	89.70% ± 0.39	92.89% ± 0.38
Decreasing learn	91.11% ± 0.12	91.36% ± 0.14
Target Selection Policy: Destroy Score		
Decreasing static	91.89% ± 0.35	93.00% ± 0.38
Decreasing learn	91.14% ± 0.12	91.39% ± 0.14
Target Selection Policy: Borda Count		
Decreasing static	89.75% ± 0.13	92.88% ± 0.38
Decreasing learn	91.25% ± 0.12	91.40% ± 0.14

Final State Prediction Accuracy

This section compares the combat models based on how well do they predict the final game state of the combat (instead of just the winner). We compared against SparCraft with the same configurations as before; and our proposed models (*TS-Lanchester*^{2}, *Sustained* and *Decreasing*) with both DPF matrices (static and learn) and all three target selection policies (Random, Destroy Score and Borda Count) as described in Section 3.2.3.

In order to assess the similarity of the predicted state by our combat models and the actual end state in the training set, we used a measure inspired in the Jaccard index (a well known similarity measure between sets: the size of their intersection divided by the size of their union). Given a combat in the training set with initial state $S = A_0 \cup B_0$, and final state $F = A_f \cup B_f$, and the final state prediction generated by a combat model $F' = A'_f \cup B'_f$, the similarity between F and F' is defined as:

$$similarity(S, F', F) = 1 - \frac{||S \cap F| - |S \cap F'||}{|S|}$$

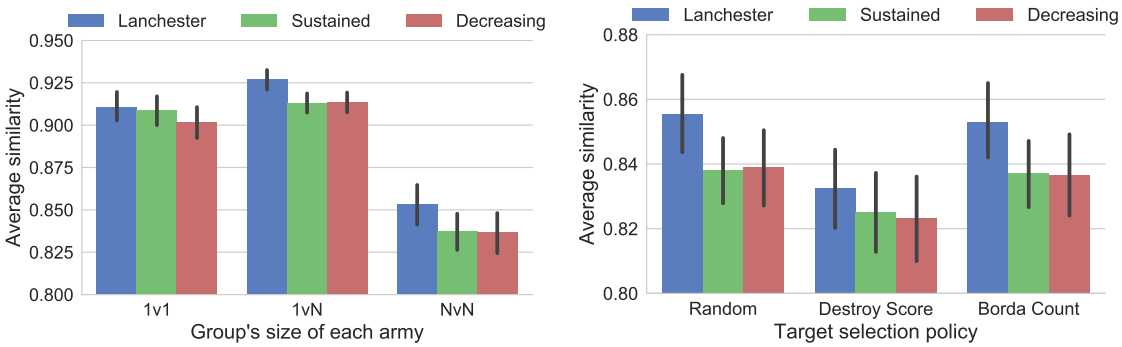
Table 3.4 shows the average similarity of the predictions generated by the different combat models with respect to the actual outcome of the combats in the training set. The best combat model in out experiments is *TS-Lanchester*^{2} with *static* DPF and a Borda Count target selection policy. Models using Borda Count are statistically significantly better than random ones according to a T-test with confidence levels lower than 0.000002. While practically there is not a difference between the performances of our proposed models (in their best configuration, which is *static* DPF and *Borda Count* target selection); our proposed models can predict the final state significantly more accurately than SparCraft. Additionally, our models can predict the final state with much lower

Table 3.4: Final state average similarity and standard error (SEM) of each combat model, and time (sec) to simulate all the combats.

Combat Model	Full Dataset	SparCraft Dataset	
	Similarity	Similarity	Time
SparCraft (AC)	N/A	0.8652 ± 0.004	0.2328
SparCraft (AW)	N/A	0.8550 ± 0.004	0.2084
SparCraft (AV)	N/A	0.8564 ± 0.004	0.2218
SparCraft (NOK-AV)	N/A	0.8592 ± 0.004	0.1871
SparCraft (KC)	N/A	0.8556 ± 0.004	0.7495
SparCraft (KV)	N/A	0.8522 ± 0.004	0.7673
Target Selection Policy: Random			
TS-Lanchester^2 static	0.9085 ± 0.002	0.9154 ± 0.003	0.0096
TS-Lanchester^2 learn	0.8767 ± 0.001	0.8853 ± 0.001	0.0070
Sustained static	0.9007 ± 0.002	0.9093 ± 0.003	0.0063
Sustained learn	0.8660 ± 0.001	0.8771 ± 0.001	0.0061
Decreasing static	0.8986 ± 0.003	0.9096 ± 0.003	0.0055
Decreasing learn	0.8892 ± 0.001	0.8924 ± 0.001	0.0060
Target Selection Policy: Destroy Score			
TS-Lanchester^2 static	0.9060 ± 0.003	0.9142 ± 0.003	0.0072
TS-Lanchester^2 learn	0.8730 ± 0.001	0.8840 ± 0.001	0.0068
Sustained static	0.9014 ± 0.002	0.9111 ± 0.003	0.0076
Sustained learn	0.8682 ± 0.001	0.8801 ± 0.001	0.0062
Decreasing static	0.9002 ± 0.002	0.9113 ± 0.003	0.0058
Decreasing learn	0.8860 ± 0.001	0.8911 ± 0.001	0.0054
Target Selection Policy: Borda Count			
TS-Lanchester^2 static	**0.9106 ± 0.002**	**0.9165 ± 0.001**	0.0069
TS-Lanchester^2 learn	0.8768 ± 0.001	0.8863 ± 0.001	0.0067
Sustained static	0.9015 ± 0.002	0.9105 ± 0.001	0.0065
Sustained learn	0.8660 ± 0.001	0.8772 ± 0.001	0.0061
Decreasing static	0.8987 ± 0.003	0.9119 ± 0.001	0.0056
Decreasing learn	0.8895 ± 0.001	0.8950 ± 0.001	0.0053

computational requirements, being between 19 to 145 times faster than SparCraft (although we'd like to point out that SparCraft was not designed for this purpose, but for performing game tree search at the scale of individual combats). This is especially important when we want to use a combat model as a forward model in the context of MCTS, where combat simulations need to be executed as part of the playouts. Although *learned* models perform slightly worse than *static* ones, the main contribution here is showing that it is possible to learn those parameters from data and do almost as well as for situations in which those parameters are known.

Moreover, we noticed that simulating combats between heterogeneous armies (armies with different types of units) are particularly hard to predict. This is shown in Figure 3.6a where the final state similarity of each model is grouped by different types of combats: between homogeneous armies (1vs1), semi-homogeneous armies (1vsN) and heterogeneous armies (NvsN) (notice that "1vs1" does not mean "one unit versus one unit", but "an army composed of all units of one type versus another army composed of units of also one type"). As expected, there is a significant difference between heterogeneous armies and the rest; while the performance with semi-homogeneous armies is slightly better than homogeneous armies due the fact that normally there is a stronger army (the one with more than one type of unit).



(a) Final state average similarity and 95% CI of each model using a Borda Count target selection policy and clustered by number of groups on each army.

(b) Final state average similarity and 95% CI of each model in combats with heterogeneous armies (NvsN) clustered by target selection policy.

Figure 3.6: Comparisons of the final state average similarity.

Figure 3.6b shows the impact on the performance of the target selection policy in combats with heterogeneous armies (NvsN). We can observe how the learned Borda Count target policy outperforms the baseline (random) and a simple heuristic (destroy score).

3.3 Adversarial Search over Abstract Game States

Let us now focus on how to incorporate the game state abstractions and forward models presented above into a concrete adversarial search approach for RTS games. More specifically the next subsections present: (1) a mapping between the low-level game state and the abstract game state; (2) a forward model for the abstract game state; (3) an abstract game state evaluation function; (4) two adversarial search algorithms to handle durative and simultaneous actions (ABCD and MCTSCD); and (5) a machine learning approach that learns probability distributions of action types from replay data in order to inform the two policies used in MCTS to guide the search.

3.3.1 Connecting Low-Level and Abstract States and Actions

In order to incorporate our MCTS approach into an actual StarCraft playing bot, we need to define a mapping between low-level states and abstract states. Moreover, our search process does not produce the exact low-level actions that the combat units need to perform, but abstract actions such as moving from one region to another, or attacking a particular region. Thus, we assume that there is a low-level agent that translates the low-level state to our abstract representation and then is capable of translating the actions defined above into actual low-level actions.

Most StarCraft bots are decomposed into several individual agents that perform different tasks in the game, such as scouting or construction [17]. One of such agents is typically in charge of combat units, and is in charge of controlling a military hierarchy architecture. The game tree search over abstract game states approach we are presenting is designed to replace such agent. Figure 3.7 shows the military hierarchy architecture used in the StarCraft bot used in our experiments [52], and where our game tree search over abstract game states is injected (*abstract layer*). As described above, our MCTS approach assigns actions to groups of units. Our StarCraft bot uses the intermediate concept of *squads* to control groups of units, which is analogous to the concept of unit abstraction representation. Unfortunately, generating the set of groups in a given low-level game state given the set of squads is not trivial, since, for example, units in a given squad might appear to split in two groups when moving from one region to another, as some units arrive earlier than others, thus breaking squad continuity. Additionally, if two squads with units of the same type arrive to the same region, they would be grouped together into a single large group according to the abstract representation, which might not be desirable.

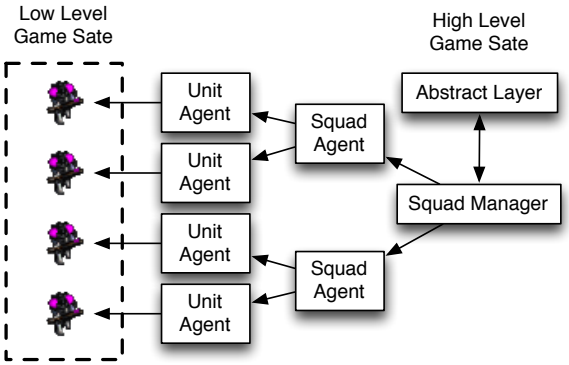


Figure 3.7: The different layers of abstraction from Low-Level Game State to High-Level Game State.

To address these problems, given a squad q in the low-level, we add each unit $u \in q$ to the abstract state; recording which abstract group the unit is being mapped to. Once all the units in the squad have been added to the abstract state, all the units are reassigned to the abstract group to which most of the units in q had been assigned. The *abstract layer* records the set of squads that were mapped to each of the abstract groups (notice that an abstract group can have more than one squad since two different squads in the same region with the same unit type will be mapped to the same group).

When simulating a combat using any of our combat models during the execution of MCTS, some units of the game state need to be removed from the simulation (since our combat models assume all units can attack and can be destroyed). First, the *invincible* and *harmless* units (i.e., the units that cannot receive or deal damage given the current armies composition) are removed from the simulation (but not from the game state). Then, the combat simulation is split in two: first we simulate the combat using only military units and after that, we simulate the combat of the winner against the *harmless* units that were removed initially from the simulation, in order to have an accurate estimate of the length of the combat.

Once the low-level state has been mapped to an abstract game state, we can use a game tree search algorithm to get an action for each group (which can be mapped to orders for each squad for execution by the other agents of the StarCraft playing bot). Since RTS games are real-time, we perform a search process periodically (every 400 game frames). After each iteration, the order of each squad is updated with the result of the search.

3.3.2 Abstract Forward Model

As we already explained, one of the particularities of RTS games is that they have *simultaneous* and *durative* actions. For this reason we need a state forwarding process that can simulate the game for as many frames as needed to reach the next decision point (a game state node where at least one player can execute an action).

Our abstract state forwarding has two components:

- **End frame action prediction**. Given a durative action (*Move*, *Attack* or *Idle*) for a group g, we predict in which game frame the action will be finished. The *Idle* action always takes 400 frames. For *Move*, we get the group velocity $g.velocity$ (all the units of the group are the same unit type) and the group location $g.location$, and target region $m.target$. Using the region distance matrix previously computed, we can apply the formula $g.velocity \times distanceMatrix[g.location][m.target]$ to get the expected end frame. For *Attack*, we use a combat model (Section 3.2) to get the minimum amount of time to finish the battle.

- **Simulation**. Then, when we reach a state where none of the players can make a move, we identify the action with the smallest *end* time and forward the game time to that moment. At

this point we need to update the outcome of the finished actions. For the *Move* action we only need to update the group position with the target position. For the *Attack* action we remove all the units that were killed during the combat simulated in our combat model (Section 3.2).

3.3.3 Abstract Game State Evaluation

A game state evaluation helps us to define how good or bad is a non terminal state (or how promising it is). In our experiments we use a simple game state evaluation where we use the *destroy score* of a unit. This score is defined by StarCraft and it is approximately $2 \times mineralCost + 4 \times gasCost$ (although some units a higher or lower score, based on their special abilities). So, given a set of friendly squads A of size n and a set of enemy squads B of size m, we calculate the following evaluation function:

$$eval(A, B) = \frac{2 \times \sum_{i=1}^{n}(A_i.size \times A_i.destroyScore)}{\sum_{i=1}^{n}(A_i.size \times A_i.destroyScore) + \sum_{j=1}^{m}(B_j.size \times B_j.destroyScore)} - 1$$

3.3.4 Alpha-Beta Considering Durations (ABCD)

To determine whether our abstraction captures enough information of the game state to produce meaningful actions, we experimented with two different game-tree search algorithms. The first one is a variant of *alpha-beta* search capable of dealing with simultaneous moves and durative actions defined by Churchill et al. [49] called ABCD (Alpha-Beta Considering Durations). They showed how this algorithm can be applied to handle low-level control for small scale combat situations in RTS games. ABCD does not automatically alternate between players, instead it determines the next player that can do a move based on the duration of the actions that are currently being executed. In game states where both players can issue actions simultaneously, ABCD defines a policy to establish the order of the player to move. In their work they experimented with two policies: RAB (Random-Alpha-Beta presented by Kovarsky and Buro [132]), that randomizes the player that will play first in each simultaneous node; and *Alt*, which alternates between players. They report better results with the *Alt* policy in their experiments, which is the one we employed in our work.

3.3.5 MCTS Considering Durations (MCTSCD)

Research exists on adapting MCTS to games with *simultaneous moves* [2, 141, 142]. In our approach we modify a standard MCTS algorithm to be able to process simultaneous and durative actions, and we call it Monte Carlo Tree Search Considering Durations (MCTSCD). Another recently defined MCTS-variant that considers simultaneous and durative actions is UCTCD [142]. But this algorithm is restricted to use an UCB policy as a tree policy. Algorithm 3 shows the pseudocode of the MCTSCD algorithm, where the simultaneous move policy is defined in PlayerToMove($state, lastSimultaneousPlayer$).

3.3.6 Experimental Evaluation

To evaluate the proposed approach, the abstraction layer is incorporated into a StarCraft bot [52] and both ABCD and MCTSCD are used to command the army during a game. The following subsections present the experimental setup and the results of the experiments.

Experimental Setup

Dealing with partial observability, due the *fog of war* in StarCraft is out of scope for these experiments. Therefore, fog of war is disabled in order to have perfect information of the game. The length of the game is limited to avoid situations where the bot is unable to win because it cannot find all the opponent's units (StarCraft ends when all the opponent units are destroyed). In the StarCraft AI competition the average game length is about 21,600 frames (15 minutes),

Algorithm 3 MCTS Considering Durations

```
1: function MctsSearch(s_0)
2:     n_0 ← CreateNode(s_0,∅)
3:     while withing computational budget do
4:         n_l ← TreePolicy(n_0)
5:         △ ← DefaultPolicy(n_l)
6:         Backup(n_l, △)
7:     return (BestChild(n_0)).action
8:
9: function CreateNode(s,n_0)
10:    n.parent ← n_0
11:    n.lastSimult ← n_0.lastSimult
12:    n.player ← PlayerToMove(s,n.lastSimult)
13:    if BothCanMove(s) then
14:        n.lastSimult ← n.player
15:    return n
16:
17: function DefaultPolicy(n)
18:    lastSimult ← n.lastSimult
19:    s ← n.s
20:    while withing computational budget do
21:        p ← PlayerToMove(s,lastSimult)
22:        if BothCanMove(s) then
23:            lastSimult ← p
24:        simulate game s with a policy and player p
25:    return s.reward
```

and usually the resources of the initial base are gone after 26,000 frames (18 minutes). Therefore, games in our experiments are limited to 20 minutes (28,800 frames). If the timeout is reached, the abstract game state evaluation is used to decide who won the game.

The experiments perform one abstract game tree search every 400 frames, and the game is paused while the search is taking place for experimentation purposes. As part of future work, we want to explore splitting the search along several game frames, instead of pausing the game.

For ABCD the following limits are defined: (1) the maximum depth is limited to 10; (2) the number of children to be considered at each node in the tree is limited to 100,000 nodes (given that the branching factor can grow enormously towards the mid game); and (3) the ABCD execution is limited to 30 seconds, i.e., if the search is taking longer than that, ABCD will start closing all the open nodes to return the best actions found so far. In the case of MCTSCD the settings are: (1) a ϵ-greedy *tree policy* with $\epsilon = 0.2$; (2) a uniform random distribution move selection for the *default policy*; (3) an *Alt* policy to decide which player will play first in a simultaneous node; and (4) MCTSCD runs for 10,000 playouts with a length of 2,880 game frames.

Three types of experiments were performed:

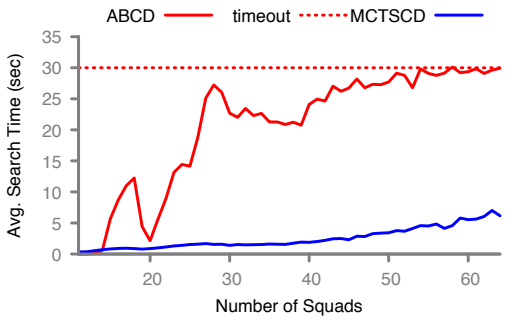
1. 100 games were run using the tournament StarCraft map Benzene to evaluate the performance of ABCD, MCTSCD and a highly optimized scripted bot called Nova [52] playing the Terran race against the built-in Terran AI of StarCraft. For this experiment we used the map abstraction **R-MA** and the forward model *Decreasing* (static DPF and the Borda Count target selection) for the *tree policy*.

2. 100 games were run using the Benzene map to evaluate each abstraction (**R-MB**, **R-MA**, **RC-MB** and **RC-MA**). Only the MCTSCD was tested playing the Terran race against the built-in Terran AI and using the forward model *Sustained* (static DPF and Borda Count target

Table 3.5: Results of two adversarial search approaches using an abstract game state representation and a scripted AI playing against the StarCraft build-in AI.

Configuration	Avg. Eval	Win %	Tie %	Loss %	Avg. Length
Scripted	0.9683	98.0	0.0	2.0	18542.5
ABCD	0.5900	16.0	83.0	1.0	26251.3
MCTSCD	0.9682	75.0	25.0	0.0	24732.1

(a) Average branching factor grouped by the number of squads in the abstract game state.



(b) Average search time for each algorithm grouped by the number of squads in the abstract game state.

Figure 3.8: Average branching factor and average search time grouped by the number of squads.

selection) for the *tree policy*.

3. 100 games were run using the Benzene map to evaluate the performance of each combat model (*TS-Lanchester*^{2}, *Sustained* and *Decreasing*; all of them using a static DPF and a Borda Count target selection) against two baselines: the original version of the Nova bot (which uses a hard-coded manually tuned strategy for controlling the military units), and a *random* approach that issues random orders (at the same level of abstraction as the MCTSCD implementation). For this experiment we used the abstraction **R-MA**.

Results

Table 3.5 shows the results of ABCD, MCSTCD and Scripted against the built-in AI. The column labeled as *Avg. Eval* shows the average value of the evaluation function at the end of the game. A tie happens when the game reaches the defined timeout and none of the players were able to destroy the other player. As reflected in *Avg. Eval*, MCTSCD achieves significantly better results than ABCD. As a reference point we compare the results against a highly optimized scripted version of the bot showing that the scripted version still achieves a higher win ratio. A close examination reveals that both ABCD and MCTSCD sometimes were not able to search deep enough in the tree to realize that they should attack the enemy, when the enemy was far away in the map. The fact that both ABCD and MCTSCD play better than the built-in AI of StarCraft indicates that the abstraction is useful in transforming the problem of large-scale combat in RTS games to a level of abstraction where game tree search is feasible and where the resulting actions are meaningful in the game.

To get further insights into the results, Figure 3.8a shows the average branching factor as a function of the number of squads in an abstract game state. As we can see, the branching factor only grows beyond 10^6 when having a large number of squads (60 or more), which, as shown in the squad frequency bars, is infrequent. In StarCraft the maximum number of units at a given time

Table 3.6: Experiments of MCTSCD with different game state abstractions using the *Sustained* combat model.

Abstraction	Avg. Eval	Win %	Loss %	Avg. Length
R-MA	**0.9391**	**60.0**	**0.0**	**24870.7**
R-MB	0.8604	25.0	4.0	25 924.2
RC-MA	0.5959	2.0	1.0	26 212.5
RC-MB	0.6139	2.0	1.0	27 561.0

Table 3.7: Experiments of MCTSCD with different forward models against two baselines.

Configuration	Avg. Eval	Win %	Loss %	Avg. Length
Scripted	0.9683	98.0	2.0	18 542.5
Random	0.4403	3.0	3.0	25 463.3
TS-Lanchester^2	0.9489	68.0	1.0	**24580.0**
Sustained	0.9391	60.0	**0.0**	24 870.7
Decreasing	**0.9682**	**75.0**	**0.0**	24 732.1

is limited to 400 (200 for each player), therefore the theoretical limit of the number of squads in our abstract representation is 400. But having 400 different type of units all spread over the map is unlikely. Hence, as we can see in Figure 3.8a the most common number of squads in the abstract representation is around 24 with an average branching factor of $10^{2.5}$.

We also analyze the amount of time spent doing search. This is a crucial point since we want to be able to execute the search during the limited time available per frame in a RTS game. Figure 3.8b shows the evolution of time spent per search by each algorithm as the number of squads in the abstract state increases. Notice how ABCD starts to saturate (stopping the search due the defined timeout) when we have more than 50 squads. After having more than 27 squads, our implementation of ABCD starts hitting the limit of 100,000 nodes explored per tree level, and thus time stabilizes. MCTSCD's time increases with the number of squads, since the time it takes the simulator to roll the game state forward increases with the number of squads. But as we can observe, the growing rate is significantly lower than the ABCD, since MCTSCD always performs the same number of playouts, regardless of the number of squads.

For each abstraction we collected the average evaluation at the end of the game, the percentage of games won (without reaching the timeout), the percentage of lost games and the average length (in frames) of the games won. In Table 3.6 we can observe how there is a large difference between using only regions or regions+chokepoints, this might be due the increment on the search space that adding new nodes (chokepoints) generates. Then, between adding or not all the buildings (*-MA vs *-MB), adding all (*-MA) seems to help win more often and sooner. A thorough inspection of the evolution of the games over time, showed that the reason is because if only the bases are added, despite it being able to kill the enemy bases in less time, it did not kill the rest of the buildings (and hence win the game) because the game tree search was not able to "see" the remaining enemy buildings. Moreover, notice that even if the win percentage seems low (e.g., 60%), the R-MA configuration did not lose any game, and in the 40% of games that timed out, our system was winning, but was just unable to find some of the last enemy units to finish the game off. This happens because some regions in the abstract representation are larger than the sight range of the units in the region, and thus, there might be unseen areas of a region unbeknownst to the MCTS algorithm.

Table 3.7 shows that *TS-Lanchester*^{2} and *Decreasing* models both outperform the *Sustained* model, with the *Decreasing* model achieving the best performance. Comparing these results against the two baselines, we can see that the *Random* approach only manages to win 3% of the games, while losing another 3%. Notice that *random* does not lose more often, since the underlying bot is still controlling unit production and micro-managing units during combat. On the other end, the

Scripted approach manages to win 98% of the games, more than our MCTSCD approach (although the *Scripted* approach lost 2% of the games, while MCTSCD with *TS-Lanchester*^{2} or *Decreasing* did not lose any games). A significant of the games were tied because, due to the too-coarse grained abstraction of the map, our MCTSCD approach did not find some of the last buildings of the enemy, thus, being unable to finish the enemy off (pointing out a clear direction for future work). Moreover, notice that although our proposed MCTSCD approach is still not at the level of performance of the *Scripted* bot, we consider this a positive result, since the *Scripted* bot NOVA has been fine-tuned over several years to play STARCRAFT, while our MCTSCD approach is non-domain specific and can be easily transferred to other games.

3.4 Informed MCTSCD

This section explores an approach to improve the performance of MCTS approaches in STARCRAFT based on modeling the behavior of human experts. Specifically, it presents a supervised probabilistic model of squad unit behavior, and shows how to train this model from human replay data. This model captures the probability with which humans perform different actions in different game circumstances. We incorporate this model into the policies of a MCTS framework for STARCRAFT to inform both the *tree policy* and the *default policy* significantly outperforming a baseline MCTS approach.

This idea has already been applied with success in other games like Poker or Go. In Poker, Ponsen et al. [143] learned an opponent model to bias the *tree policy*. Coulom [144] calculated the Elo rating in Go to inform both policies (tree and default), Gelly and Silver [145] experimented with combining offline reinforcement learning knowledge and online sample-base search knowledge; this last idea was further explored later in AlphaGo [18]. In our previous work, we showed this can also improve small-scale RTS gameplay [146]. In the following subsections, we show how this idea can be scaled up to large RTS games, such as STARCRAFT, and how can replay data be used to generate the required training set.

Squad-Action Naive Bayes Model

This model captures the probability distribution by which human experts perform each of the given actions in a given situation and given the actions that are legal for a given squad. As defined in Section 3.1.3, the set of all possible action types T a squad can perform (*Idle*, *Attack* and *Move* (toFriend, toEnemy, towardsFriend, towardsEnemy) contains 18 different action types (we distinguish *Move* actions by their features (described above), since there are 2^4 different *Move* actions, the total number of different action types a squad can perform is $2 + 2^4 = 18$). Moreover, in a given game situation the set of actions a squad can perform is bounded by the number of adjacent regions to the region at hand, and there might be more than one squad action with the same action type (e.g., there might be more than one adjacent region characterized by the same move features). Let us define $\mathcal{T} = \{t_1, ..., t_{18}\}$ to be the set of action types that squads can perform, and let $\mathcal{A}_s = \{a_1, ..., a_n\}$ to be the set of squad actions a squad can perform in a given game state s (we call this the set of *legal* actions), where we write the type of an action as $type(a) \in T$. Now, let $X_1, ..., X_{18}$ be a set of Boolean variables, where X_i represents whether in the current state s, the squad at hand can perform an action of type t_i. Moreover, variable T denotes the type of the action selected by the squad in the given state ($T \in \mathcal{T}$), and A denotes the actual action a squad will select ($A \in \mathcal{A}_s$). Then, we will use the conditional independence assumption, usually made by Naive Bayes classifiers, that each variable X_j is conditionally independent of any other variable X_i given T, to obtain the following model:

$$P(T|X_1, \ldots, X_n) = \frac{1}{Z} P(T) \prod_{j=1}^{n} P(X_j|T)$$

where Z is a normalization factor, $P(T = t_i)$ is the probability of a squad to choose action of type t_i given that $X_i = true$, and $P(X_j|T)$ is the probability distribution of feature X_j, given T. Below

we will show how to estimate these probability distributions from replay data.

Finally, since there might be more than one action in $\mathcal{A}_s$ with the same given action type (e.g., there might be two regions a squad can move to characterized by the same features), we calculate the probability of each action as:

$$P(A|X_1,\ldots,X_n) = \frac{P(type(A)|X_1,\ldots,X_n)}{|\{a \in \mathcal{A}_s|type(a) = type(A)\}|}$$

We incorporate the model described above into MCTS. As described above, MCTS employs two different *policies* to guide the search: a *tree policy* and a *default policy*. The squad-action probability model can be used in MCTS in both policies.

Moreover, while a squad-action probability model can be used directly as a *default policy*, to be used as a *tree policy*, it needs to be incorporated into a multi-armed bandit policy.

Informed ϵ-Greedy Sampling

The *tree policy* of MCTS algorithms is usually defined as a *multi-armed bandit* (MAB) policy. A MAB is a problem where, given a predefined set of actions, an agent needs to select which actions to play, and in which sequence, in order to maximize the sum of rewards obtained when performing those actions. The agent has no information of the expected reward of each action initially, and needs to discover them by iteratively trying different actions. MAB policies are algorithms that tell the agent which action to select next, by balancing *exploration* (when to select new actions) and *exploitation* (when to re-visit actions that had already been tried in the past and looked promising).

MAB sampling policies traditionally assume that no priori knowledge about how good each of the actions exists. For example, UCT [147], one of the most common MCTS variants, uses the UCB1 [148] sampling policy, which assumes no a priori knowledge about the actions. A key idea used in AlphaGO is to employ a MAB policy that incorporated a prior distribution over the actions into a UCB1-style policy. Here, we apply the same idea to ϵ-greedy.

As any MAB policy, *Informed ϵ-greedy sampling* will be called many iterations in a row. At each iteration t, an action $a_t \in \mathcal{A}$ is selected, and a reward r_t is observed.

Given $0 \leq \epsilon \leq 1$, a finite set of actions A to choose from, and a probability distribution P, where $P(a)$ is the a priori probability that a is the action an expert would choose, Informed ϵ-greedy works as follows:

- Let us call $\overline{r}_t(a)$ to the current estimation (at iteration t) of the expected reward of a (i.e., the average of all the rewards obtained in the subset of iterations from 0 to $t-1$ where a was selected). By convention, when an action has not been selected before t we will have $\overline{r}_t(a) = 0$.
- At each iteration t, action a_t is chosen as follows:
  - With probability ϵ, choose a_t according to the probability distribution P.
  - With probability $1-\epsilon$, choose the best action so far: $a_t = argmax_{a \in A}\overline{r}_t(a)$ (ties resolved randomly).

When P is the uniform distribution, this is equivalent to the standard ϵ-greedy policy. We will use the acronym *NB-ϵ* to denote the specific instantiation of informed ϵ-greedy when using our proposed Naive Bayes model to generate the probability distribution P.

Best Informed ϵ-Greedy Sampling

Best Informed ϵ-Greedy Sampling is a modification over the previous MAB policy, that treats the very first iteration of the MAB as a special case, Specifically, at each iteration t, action a_t is chosen as follows:

- If $t = 0$, choose the most probable action a_t given the probability distribution P: $a_t = argmax_{a \in A}P(a)$.

- Else, use regular Informed ϵ-Greedy Sampling.

Our experiments show that this alternative MAB policy is useful in cases where we have a small computation budget (e.g., in the deeper depths of the MCTS tree), and thus, seems appropriate to real-time games.

We will use the acronym *BestNB-ϵ* to denote the specific instantiation of best informed ϵ-greedy when using our proposed Naive Bayes model to generate the distribution P.

Other MAB Sampling Policies

The idea of incorporating predictors (probabilistic or not) has been explored in the past in the context of many other MAB sampling policies. For example, PUCB (Predictor + UCB) [149] is a modification of the standard UCB1 policy incorporating weights for each action as provided by an external predictor (such as the one proposed in this paper). Chaslot et al. [150] proposed two *progressive strategies* that incorporate a heuristic function over the set of actions that is taken into account during sampling. Another example is the sampling policy used by AlphaGO [18], which is related to both progressive strategies.

One problem of UCB-based policies is that they require sampling each action at least once. In our setting, however, there might be nodes in the MCTS tree with a branching factor larger than the computational budget, making those policies inapplicable. An exception is PUCB, which is designed for not having to sample each action once. We experimented with PUCB in our application domain with poor results. However, since even if PUCB does not require to sample all the actions once, it still requires to reevaluate the value of each action in order to find the action that maximizes this value. Given the large branching factor in our domain, this resulted in impractical execution times, which made us only consider ϵ-greedy-based strategies. As explained below, as part of our future work, we would like to explore additional policies taking into account the particularities of our domain.

3.5 Adversarial Search Experiments

In order to evaluate the performance of informed MCTSCD we performed a set of experiments using our StarCraft bot (Nova [52]) that uses the proposed informed MCTSCD to command the army during a real game.

Training Data

To be able to learn the parameters required by the squad-action Naive Bayes model, we need a dataset of squad actions. We extracted this information from professional human replays. Extracting information from replay logs has been done previously [29, 134, 139]. To generate the required dataset we built a parser that, for each replay, proceeds as follows:[4]

- First, at each time instant of the replay, all units with the same type in the same region are grouped into a squad.
- Second, for each region with a squad, the set of legal actions is computed.
- Third, the action that each unit is performing is transformed to one of the proposed high-level actions (if possible). For example, low-level actions like *Move*, *EnterTransport*, *Follow*, *ResetCollision* or *Patrol* become *Move*. In the case of the high-level action *Move*, the target region is analyzed to add the region features to the action.
- Fourth, the most frequent action in each squad is considered to be the action that squad is executing.

[4]Source code of our replay analyzer, along with our dataset extracted from replays, can be found at `https://bitbucket.org/auriarte/bwrepdump`

- Finally, for each squad and each time instant in a replay, we compute its squad state as $s = (g, r, t, T, A)$ where g is the unit's type of the squad, r is the current region of the squad, t is the type of the action selected, T is the set of types of the legal actions at region r, and A is the actual set of legal actions at region r.

Each time that the state of a squad changes in a replay (or a new squad is created), we record it in our dataset, and it constitutes one of the training instances.

Once we have the training set $S = \{s_1, \ldots, s_m\}$, we can use the Maximum Likelihood Estimation (MLE) to estimate the parameters of our model as:

$$P(T = t) = \frac{|\{s \in S | s.t = t\}|}{|\{s \in S | t \in s.T\}|}$$
$$P(X_j = true | T = t) = \frac{|\{s \in S | s.t = t \wedge t_j \in s.T\}|}{|\{s \in S | s.t = t\}|}$$

Experimental Setup

Dealing with partial observability is a complex problem that we address in the next section (Section 3.6). In order to simplify the experiments in this section we decided to disable the fog-of-war to have perfect information of the game. We also limited the length of a game to avoid situations where bots are unable to win because they cannot find all the opponent's units (StarCraft ends when all the opponent's buildings are destroyed). In the StarCraft AI competition the average game length is about 21,600 frames (15 minutes), and usually the resources of the initial base are gone after 26,000 frames (18 minutes). Therefore, we decided to limit the games to 20 minutes (28,800 frames). If we reach the timeout we consider the game a tie.

In our experiments, we performed one call to informed MCTSCD every 400 frames, and pause the game while the search is taking place for experimentation purposes.

Informed MCTSCD has several parameters which we set as follows: for any policy using an ϵ-greedy component ϵ is set to 0.2; to decide the player in a *simultaneous node* we use an *Alt* policy [49] that alternate players; the *length of playouts* (or simulations) is limited to unfold until 2,880 frames (2 minutes of gameplay; this number is extracted from an empirical evaluation described in the next subsection) or until a terminal node is reached (and with a general timeout of 28,800 frames); and as a forward model for playouts (or "simulator") we use the *Decreasing DPF model* (where the DPF of an army is decreased every time a unit is killed) using a learned Borda Count target selection policy [151]. We experimented with executing informed MCTSCD with a computational budget from 1 to 10,000 playouts; and with the following configurations of (*tree policy*, *default policy*):

- **(ϵ, UNIFORM)**. Our baseline using an ϵ-greedy for the tree policy, and a uniform random default policy.
- **(ϵ, NB)**. Same as previous but changing the default policy to our proposed *Squad-Action Naive Bayes Model*.
- **(NB-ϵ, NB)**. An *informed ϵ-greedy sampling*, that uses our proposed *Squad-Action Naive Bayes Model* to generate the probability distribution P, for the tree policy.
- **(BestNB-ϵ, NB)**. For this configuration we changed the tree policy to use a *best informed ϵ-greedy sampling* with the *Squad-Action Naive Bayes Model*.

The previous configuration is summarized in Figure 3.9. We used the StarCraft tournament map Benzene for our evaluation and we ran 100 games with our bot playing the Terran race against the built-in Terran AI of StarCraft that has several scripted behaviors chosen randomly at the beginning of the game.

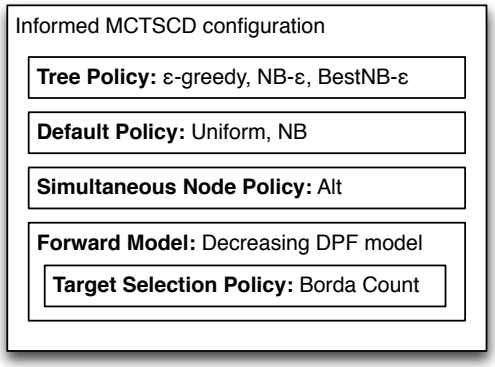


Figure 3.9: Experiment configuration of Informed MCTSCD.

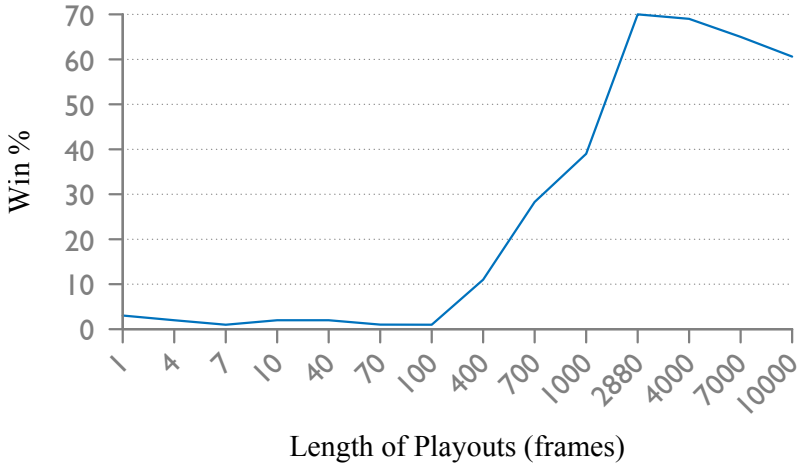


Figure 3.10: Win % using a MCTSCD with an ϵ-greedy tree policy and a uniform random default policy.

Results

In our first experiment we evaluated the performance of MCTSCD with *playouts* (or simulations) of different durations. The computational budget of MCTSCD was fixed to 1,000 playouts, and we used standard ϵ-greedy for the tree policy and a uniform distribution as the default policy. Intuitively, increasing the length of playouts increases the lookahead of the search. As we can observe in Figure 3.10, performance is very low if playout length is kept below 100 frames. It increases exponentially between 100 up to 2,880 frames (from $\approx$ 4 seconds to 2 minutes), and after that the performance start to degrade. Our hypothesis concerning the performance degradation after 2,880 frames is because of the inaccuracies in our forward model. For example, the forward model used, does not simulate production, so after 2 minutes of playout simulation, the resulting state would be probably different from the actual state the game will be in after 2 minutes given that no new units are spawned during playouts. Secondly, our forward model is only an approximation, and the slight inaccuracies in each simulation over a chain of approximate combat simulations can compound to result into inaccurate simulations.

Figure 3.11 shows the win % (i.e., wins without reaching the timeout) using several tree and default policies for different computation budgets. Starting from the extreme case of running MCTSCD with a single playout (which basically would make the agent just play according to the tree policy, since the first child selected of the root node will be the only one in the tree, and thus the move

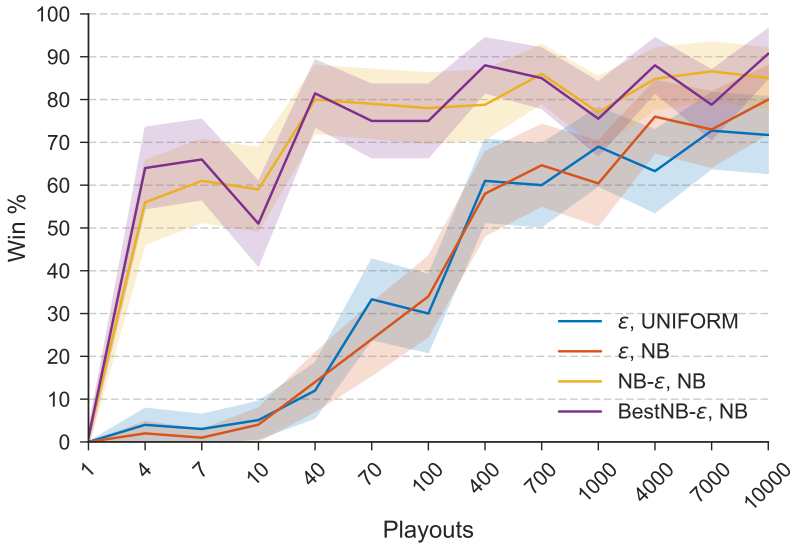
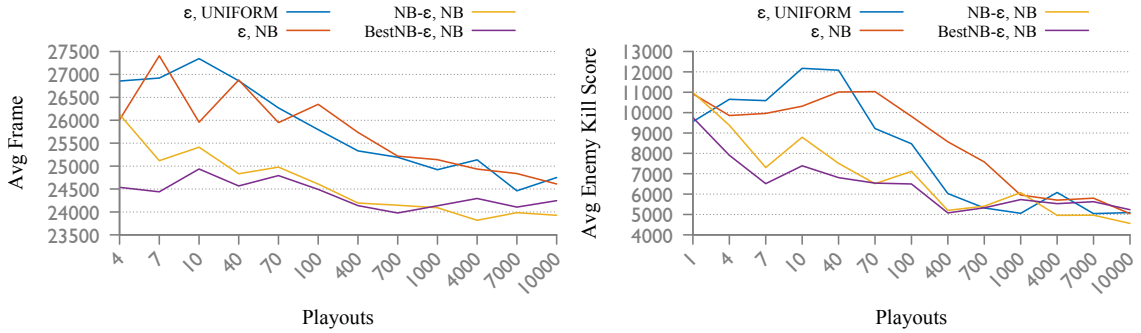


Figure 3.11: Comparison of Win % and 95% CI using a MCTSCD with different policies (tree policy, default policy).



(a) Comparison of average frames it took to win a game using a MCTSCD with different policies (tree policy, default policy). Lower is better.

(b) Comparison of average enemy's kill score using a MCTSCD with different policies (tree policy, default policy). Lower is better.

Figure 3.12: Comparison of average frames and average enemy's kill score.

to be played), all the way to running 10,000 playouts. Playout length was set to 2,880 frames. As expected, as the number of playouts increases, performance also increases. Moreover, we can observe a significant difference between using a standard ϵ-greedy tree policy (ϵ), which achieves a win % of about 70% when using 10,000 playouts, and using an informed tree policy (BestNB-ϵ or NB-ϵ), which achieve a win % of 90% and 85% respectively. Additionally, using informed tree policies, we reach a win % of about 80% by using as few as just 40 playouts (which is a surprisingly low number!). This shows that the probability distribution captured by the Naive Bayes model can significantly help MCTSCD during the search process in guiding the search toward promising areas of the search tree. On the other hand, the performance difference between using an informed default policy or not (*NB* vs *UNIFORM*) is not large.

Figure 3.12a shows the average amount of time (in game frames) that our approach took to defeat the opponent, showing again a clear advantage for informed policies. Figure 3.12b shows the average *kill score* achieved by the opponent at the end of the game. Since this captures how many units our MCTSCD approach lost, lower is better. Again we see a clear advantage of informed strategies, but this time only for small number of playouts.

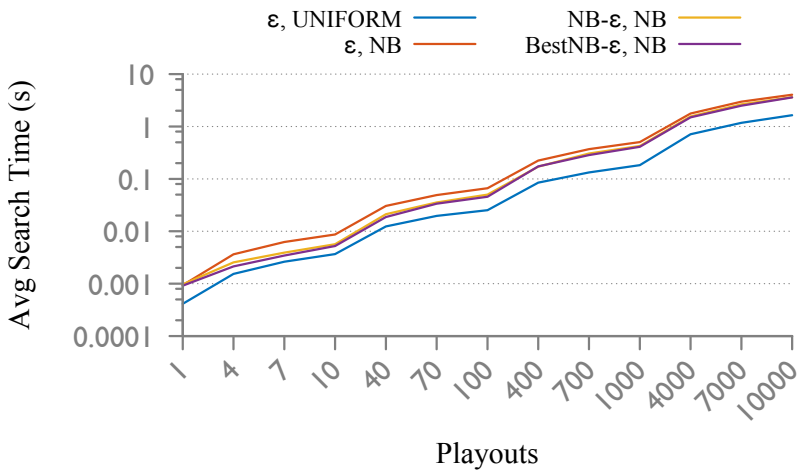
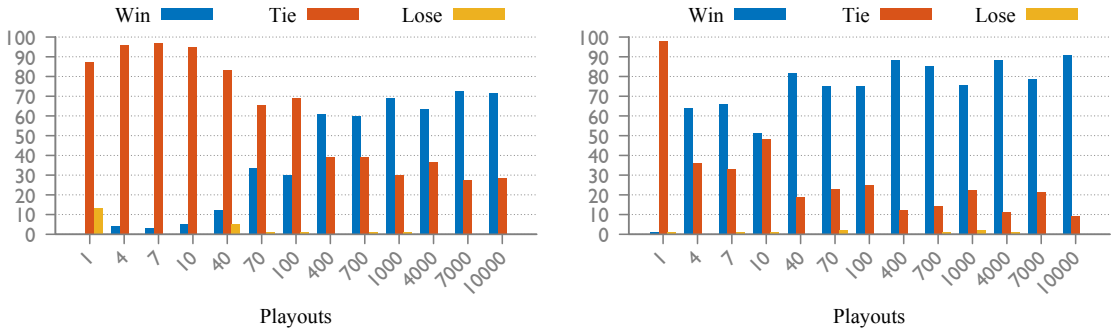


Figure 3.13: Comparison of average search time using MCTSCD with different tree and default policies.



(a) Win/Tie/Lose % of MCTSCD(ϵ,UNIFORM). **(b)** Win/Tie/Lose % of MCTSCD(BestNB-ϵ,NB).

Figure 3.14: Win/Tie/Lose % of different MCTSCD policies.

Finally, we analyzed the computation time required by each configuration, since we are targeting a real-time environment. Figure 3.13 shows how the increment of playouts leads to a linear time growth, and it shows that the *UNIFORM* default policy is faster. Mainly due the fact that it has lower chances to engage combats, which are expensive to simulate using our forward model (this is because, when there is a combat, our forward model needs to simulate the attacks of all the units involved, which requires more CPU time than when units just move around in squads).

In summary, we can see that adding the Naive Bayes model learned offline into MCTSCD improves the performance significantly. Of particular interest for RTS games is the fact that performance is very good with a small number of playouts, since the model can guide MCTS down the branches that are most likely to be good moves. Using less than 100 playouts in any of our informed scenarios is enough to match the performance of MCTSCD with 10,000 playouts. Notice also that the search time with 100 playouts is less than 0.1 seconds. Suggesting that this approach could be applied without pausing the game during the searches. Moreover, we would like to point out (as shown in Figures 3.14a and 3.14b) that the remaining 10% - 15% of games that are not shown as wins in Figure 3.11 for BestNB-ϵ or NB-ϵ, are actually not loses, but ties, and most of those are ties because our system defeated the enemy but was unable to find the last buildings. This was due to a limitation in our abstraction, where if a region is too large, MCTSCD does not have the capability of asking a unit to explore the whole region (since for MCTSCD that whole region is a single node in the map graph). As part of our future work, we would like to improve our map abstraction for

not including regions that are larger than the average visibility range of units, in order to prevent this from happening.

3.6 Handling Partial Observability

Games with imperfect information are usually exponentially harder than games with perfect information. Two-player unbounded-length games where players have perfect information, i.e., those that can be modeled using *alternating Turing machine* (ATM) such as Chess, are **EXPTIME** [10]; while if there is private information, i.e., games that require *private alternating Turing machine* (PATM) such as *PRIVATE-PEEK*, they are **2-EXPTIME** [13]. Therefore, any attempt to try to find an optimal strategy will be intractable. This section approaches the problem of solving games with imperfect information and large branching factors by generating a **single believe state** as close as possible to the real game state that can be used as the starting point for search by any game tree search algorithm, instead of the partially observed game state (i.e., doing *determinization*).

Parker et al. [109] proposed 4 different sampling strategies for large information sets, but they require to perform depth-first search over the space of observations, something that could be expensive in domains with time constrains. Richards and Amir [152] improved the performance of the previous approach by modeling the problem as a Constrain Satisfaction Problem (CSP). The approach we present here, instead of generating a pool of valid believe states, tries to generate a single believe state as close as possible to the real game state. The hypothesis is that since the pace of RTS games is very fast, we do not need to find the optimal strategy every frame, therefore sampling a game state that approximates the actual game state sufficiently well will be enough to converge towards a strong gameplay.

In this section, we assume that both players know the initial board configuration, i.e., the initial state is fully observable. This assumption is true for board games like Kriegspiel where the initial board configuration is known for both players, or for RTS games where for a given map the initial base locations are known for both players (like in μRTS). For other games like Poker this is not true since the opponent hand is unknown, but those are out of the scope of this work. With this assumption we propose three different strategies for sampling a single state, which we call the *believe state* from the current information set:

- **Goal Seeker** (GS): Given an initial state s_0 with perfect information, this strategy records the location of each known opponent unit that cannot move (e.g., buildings), U_{nm}. Then, in order to generate a believe state s_b from the current observation z, all the units in U_{nm} are added to z if they are not already there. If a unit present in the memory is destroyed from the game state, then it is removed from the memory as well. This can be seen as an extreme version of the *overconfidence* player model from [110], where the opponent always chooses to do nothing. We call this strategy the "goal seeker", since it has the effect of just remembering where the opponent base is (since this is known at the start of the game), and thus, tends to go toward the enemy base right away.

- **Imperfect Memory** (IM): Given a fully observable initial state s_0, this strategy records the location of **all** opponent units that we observed at some point, but that are not currently visible into a record U_o. Then, given a current observation z, if the location where a unit in U_o is visible but the unit is not there anymore, the unit is removed from U_o. This method basically adds to the current observation the "last known enemy unit location" of all the units that we cannot currently see, but that were observed at some point. We call this "imperfect memory" since if we saw a unit in a position x, which then became unobservable, if we observe x again, but there is no unit there, then we forget that we ever was a unit there.

- **Perfect Memory** (PM): This strategy acts like IM but adds an inference mechanism. The inference mechanism is used in two situations: 1) when the location of a unit in U_o is not visible, it updates the unit's location in U_o with the closest not observable location from the unit's location in the memory (i.e., it assumes the unit has moved, but just the minimum amount of move as for making the unit not observable); and 2) it adds units that we have

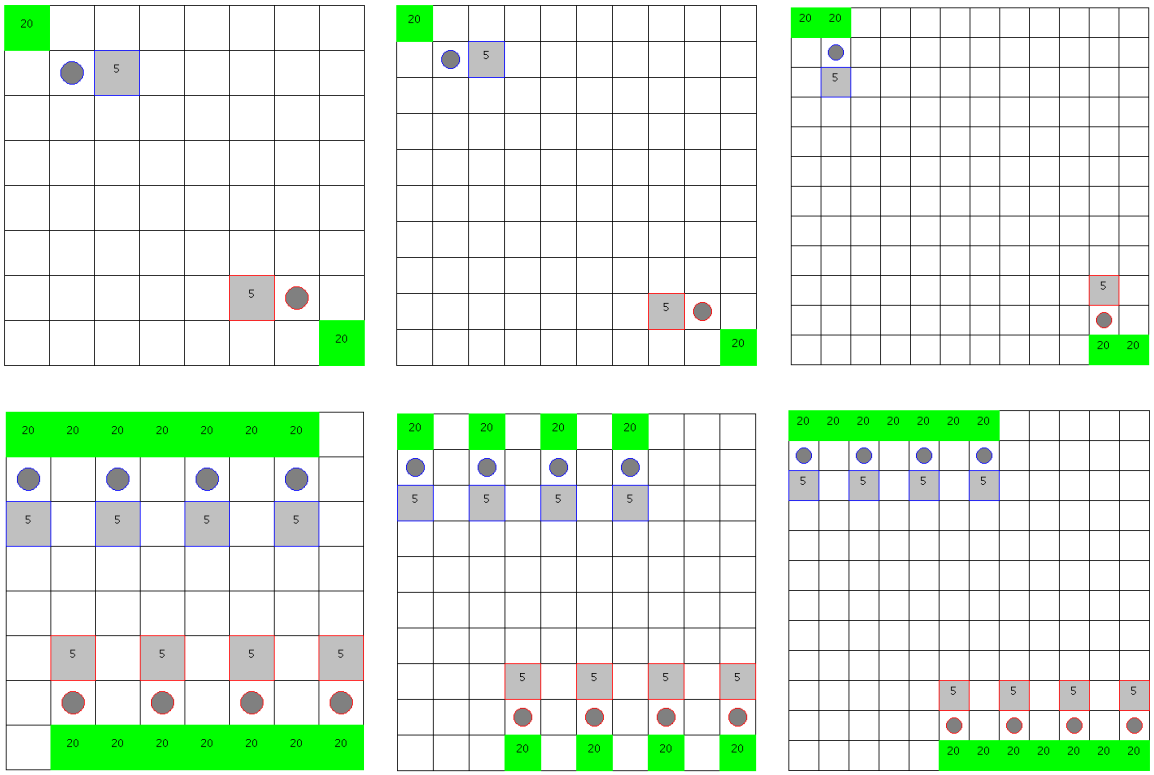


Figure 3.15: μRTS maps used in our experiments: 1BW8x8 (top left), 1BW10x10 (top center), 1BW12x12(top right), 4BW8x8 (bottom left), 4BW10x10 (bottom center), and 4BW12x12 (bottom right).

never seen but they are required to explain part of the observation (i.e., if in order for the opponent to have units of a certain type t_1, the opponent must have first build a unit of type t_2, if we observe a unit of type t_1, then for sure we know the opponent has a unit of type t_2). Inferred units are added in the closest non-observable location to the opponent's unit that caused the inference. In other words, this sampling never forgets a unit that it has seen (hence the name "perfect memory"), it also tries to guess their location, and it infers units that cannot be seen but that must be there.

From a point of view of game theory, these samplings capture the *memory of past knowledge* [153] at different degrees of accuracy.

3.6.1 Experimental Evaluation

For this experiment, we first used μRTS to have a more fine grained control of the different game properties and smaller scenarios to test the scalability. Then we tested the best configuration in StarCraft to evaluate the performance in a more complex domain.

We ran the μRTS experiments in six different maps (Figure 3.15): three standard game maps where each player starts with one base and one worker but with different map sizes (1BW8x8, 1BW10x10, 1BW12x12) and three maps where each player starts with four bases and four workers (4BW8x8, 4BW10x10, 4BW12x12). Given the small size of the maps being used for evaluation, the default visibility ranges of some of the units in μRTS were too large (basically the whole map was almost visible at the start in 8x8 maps). So, we reduced the visibility range of bases to 3, and of workers to 2. In our experiments, we used the following AIs:

- *RandomBiased*: It selects one of the possible player-actions at random, but with 5 times more probability of selecting an attack or a harvest action than any other action.
- *Partial Observability Worker Rush (POWorkerRush)*: It is a hard-coded strategy that constantly produces "Workers" to attack the nearest target or if there are not enemies in sight, they move to the nearest not visible location. It only uses one worker to mine resources.
- *Partial Observability Light Rush (POLightRush)*: Like *POWorkerRush* but it builds a barracks, and then constantly produces "Light" military units instead of "Workers".
- ϵ-*Greedy MCTS* (ϵ-MCTS): As a baseline for game tree search we selected the well known MCTS algorithm with an ϵ-greedy sampling strategy ($\epsilon = 0.25$ as shown to be best in [2]) as a tree policy, and a *RandomBiased* default policy to simulate 200 game cycles. The evaluation function used was: the sum of the cost in resources of all the player units in the board weighted by the square root of the fraction of hit-points left, then subtract the same sum for the opponent player. We tested with different number of playouts per cycle (100, 500 and 1,000). Although there are better game tree search algorithms for RTS games like Portfolio Greedy Search [142], LSI [154] NaiveMCTS [2] or Informed MCTS [120, 146] we wanted to test the performance of a partially observable game state with a simple and well known game tree search algorithm since the performance difference between the different single believe state sampling strategies should be analogous for other search algorithms. This AI uses directly the partially observable state, i.e., if there is no enemies in the partially observable state the AI thinks it won and it might return an arbitrary action.
- *GS* ϵ-*MCTS*: Like ϵ-MCTS AI but at each frame the partially observable game state is enhanced with the Goal Seeker sampling.
- *IM* ϵ-*MCTS*: Like ϵ-MCTS AI but at each frame the partially observable game state is enhanced with the Imperfect Memory sampling.
- *PM* ϵ-*MCTS*: Like ϵ-MCTS AI but at each frame the partially observable game state is enhanced with the Perfect Memory sampling.
- *Cheating* ϵ-*MCTS*: Like ϵ-MCTS AI but this time instead of a partially observable state we receive a fully-observable game state. Notice that this amounts to cheating, since the opponent still receives a partially-observable game state. The purpose of using this AI is to offer an upper bound of the performance that can be expected out of ϵ-MCTS.

For each pair of AIs (3 hard-coded and 5 MCTS, with 3 different playout budgets each, results in 18 different AIs, and a total of $\binom{18}{2} = 153$ match-ups), we ran 20 games per map and per match-up (each AI plays 20 games as player 1 and another 20 games as player 2 in the same map against the same opponent) in 6 different maps, resulting in a total of $153 \times 20 \times 6 = 18360$ games. We limited each game to 3000 cycles (5 minutes), after which we considered the game a tie.

For the experiments in StarCraft we selected the best algorithm of the previous section: Informed MCTSCD using a *best informed* ϵ-*greedy sampling* ($\epsilon = 0.2$) for the tree policy; a *Squad-Action Naive Bayes Model* for the default policy; an *Alt* policy in nodes where both players can move; a limit of 2,880 frames for the length of playouts (or simulations); a general timeout of 28,800 frames; and a *Decreasing DPF model* for the forward model. We experimented with executing informed MCTSCD with a computational budget from 1 to 10,000 playouts; and for each budget we run 100 games between informed MCTSCD with full observability (cheating) playing Terran against the built-in AI playing Terran and another 100 games between informed MCTSCD with partial observability using a *Perfect Memory* believe state sampling against the built-in AI. All games were performed in the Benzene StarCraft tournament map.

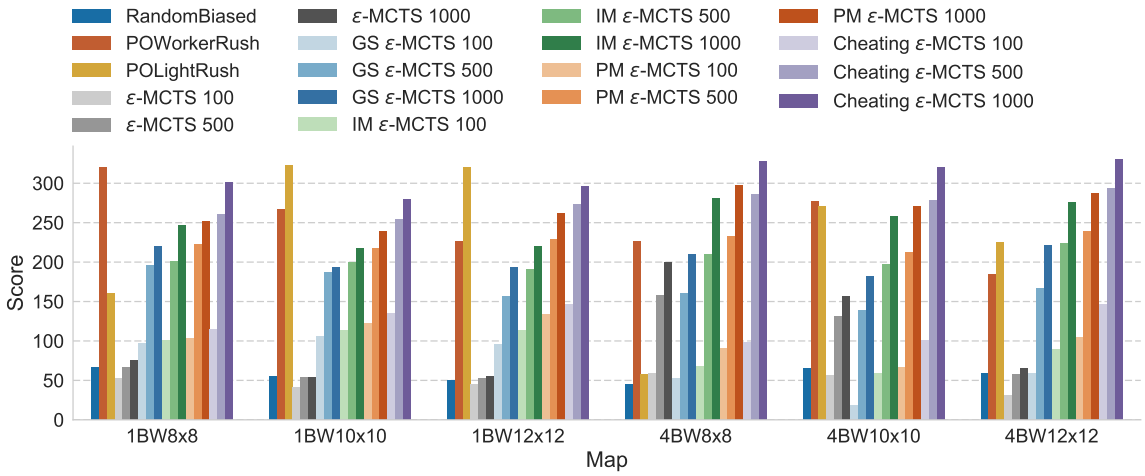


Figure 3.16: Accumulated score (wins + 0.5 × ties) obtained by each AI in each of the different μRTS maps (maximum score would be 340, meaning winning every single game).

Results

Figure 3.16 shows the summarized results of the μRTS experiments. For each map and for each AI, we show a "score", calculated as $wins + 0.5 \times ties$. From this Figure 3.16 we can make the following observations:

- **Hard-coded strategies**: These strategies perform very well in maps with a single base and worker, since that's the scenario for which they were designed. The main drawback of hard-coded strategies is that they cannot handle all type of situations. This can be seen in maps with 4 bases where *POWorkerRush* and *POLightRush* do not know how to exploit all the bases and they perform poorly against MCTS. We can also observe how the performance of *POWorkerRush* decreases by the size of the map, while *POLightRush*'s performance increases. This shows how these strategies lack generality (they perform very well in some situations, outperforming all other techniques, but they under-perform in others).

- **Number of playouts**: As expected, the performance of MCTS based AIs increases if we perform more playouts (100, 500 and 1000 were tested in our experiments). Showing a large improvement between 100 and 500 but a small one between 500 and 1000. For the purposes of our evaluation we fixed the number of playouts for comparison. But keep in mind that for RTS games the budget is usually limited by time and not by playouts, therefore MCTS AIs that can perform more playouts in the same amount of time will perform in practice better than other theoretically stronger MCTS AIs that need more time to finish a playout.

- **Size of non-observable map positions**: The partially observable game state property in RTS games comes from the fog-of-war that establishes that we cannot observe the parts of the map that are out of sight of our units. Therefore if we are able to position all our units to cover all the map, the game state becomes full observable. This property can be seen in 4BW8x8 map where we already begin with spread units that can cover almost the entire small map. Hence the good performance of ϵ-*MCTS* in the 4BW8x8 map.

- **Single believe state generation performance**: As we can see in Figure 3.16, even the simple GS sampling is better than not using any believe state estimation at all. IM is better than GS and PM is better than IM ($none < GS < IM < PM < cheating$). Moreover, we can see that the performance of PM is very close to that of cheating in most scenarios (the largest difference is seen in the 4BW10x10 map).

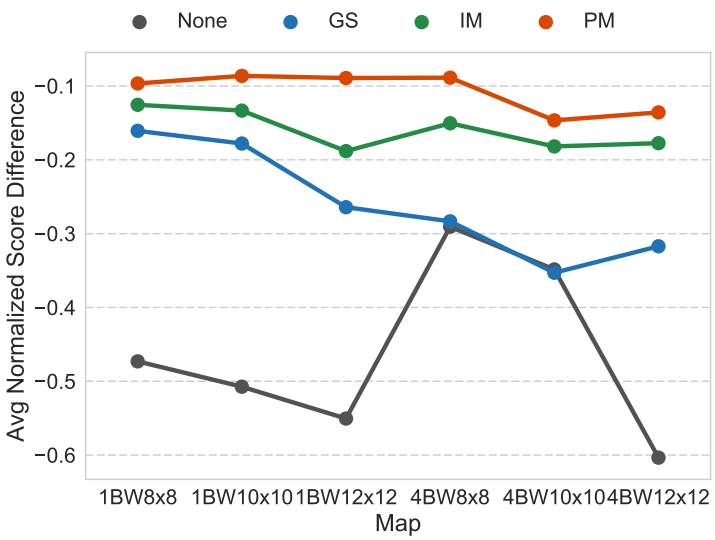


Figure 3.17: Avg normalized score difference of each believe state sampling respect cheating.

Figure 3.17 shows a deeper analysis of how accurate are the believe state estimations made by GS, IM and PM by computing the average normalized score ($wins + 0.5 \times ties$) difference for each believe state sampling against cheating (perfect information). It reveals that PM only suppose a 8%-15% decrease in performance with respect to cheating, while without any sampling we see performance penalties of up to 60%. Another observation is that while the performance penalty of PM is more constant across the maps, the performance of the other believe state samplings is more sensitive to the map size.

Analyzing the game replays we also observed that PM exhibits and emergent behavior of units trying to find a previously seen enemy. However this is an illusion of gathering information or scouting since the game tree search believes it has perfect information and tries to attack the unit in the believe state, and when it approaches the believe state keeps relocating the position of the unit in the believe state. But this is also the key of the higher performance of PM since it will search for remaining opponent units in the map (since it knows they are there), while other sampling strategies will stop searching and hence ending more games in a tie.

We also analyzed the similarity between the generated believe state and the real game state. We used a measure inspired in the Jaccard index (a well known similarity measure between sets: the size of their intersection divided by the size of their union). Given the full observable game state s (s is a set of units), a believe state s_p from the point of view of player p, and given the opponent player q, the similarity between s and s_p in estimating the units of q is defined as:

$$J(q, s, s_p) = \frac{unitDist(q, s, s_p)}{units(q, s_p) + units(q, s) - unitDist(q, s, s_p)}$$

Where $units(q, s)$ is the number of units of player q in s, and $unitDist(q, s, s_p)$ is the sum of normalized distances between the units of player q that exist both in s and s_p. This measure gives us the notion of how well are we guessing the opponent's units. Figure 3.18 shows the average Jaccard index over time and the 95% confidence interval (CI) for each believe state sampling and for each map. As we can see PM outperforms the other believe state sampling strategies by a large margin. Notice how for maps bigger than 8×8 the prediction accuracy of all samplings starts to decrease quickly at the beginning of the game until combat is engaged (usually at frame 250-300) to increase again until frame 600-750 and finally decrease slowly until the end. To avoid this "depression" at the beginning of the game, human players usually send a scout to the enemy's base to gather information. Also, small maps or maps with units covering a good portion of the map

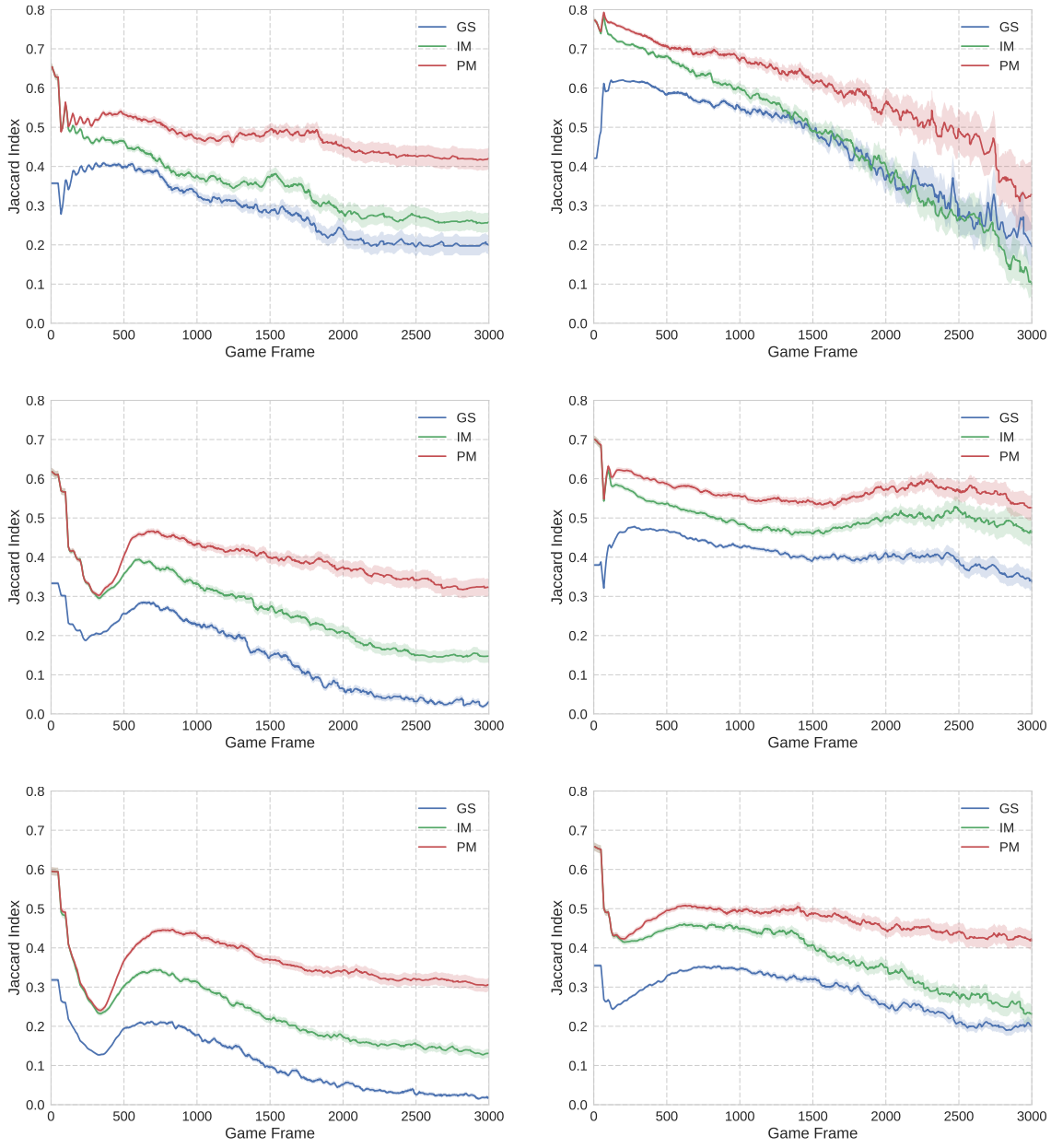


Figure 3.18: Average Jaccard index between the believe state and the real game state (1 means perfect matching, 0 means nothing in common) over time (game frames) and 95% CI. Left column are the maps with one base: 1BW8x8 (top left), 1BW10x10 (middle left) and 1BW12x12(bottom left); right column are maps with four bases: 4BW8x8 (top right), 4BW10x10 (middle right), and 4BW12x12 (bottom right).

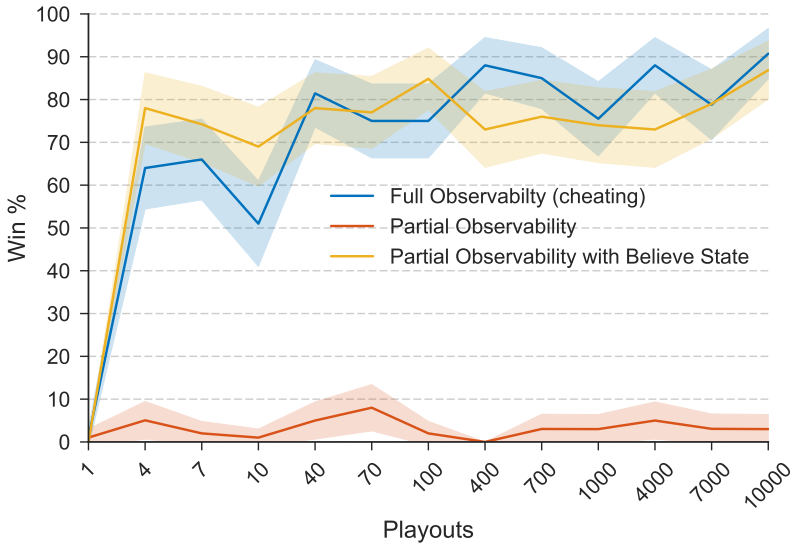


Figure 3.19: Comparison of Win % and 95% CI using informed MCTSCD with perfect information (cheating) or imperfect information with and without a *Perfect Memory single believe state* against the StarCraft built-in AI.

from the initial state (1BW8x8, 4BW8x8, 4BW10x10) do not exhibit the "depression", meaning that from the beginning we almost have perfect information and we keep losing information over time (specially for the losing player). The fact that the 95% CI grows over time is because we have less samples of long games, especially for small maps.

Figure 3.19 shows the win % average and the 95% confidence interval of informed MCSTCD using perfect information (cheating) or using imperfect information (with and without the proposed *Perfect Memory single believe state* approach) in StarCraft. As expected, when we enable the fog-of-war, informed MCTSCD algorithm without a believe state is not aware of the enemy and it loses most of the games. But once we use the *Perfect Memory single believe state* approach, the win % is similar than when we have perfect information of the game. Although the results seem to show that with less playouts the *single believe state* performs slightly better than the cheating version, and the opposite for a large number of playouts, the differences are not statistically significant. Thus, we can conclude that the proposed *Perfect Memory* believe state estimation approach seems to be enough to bring the performance of the StarCraft playing bot used in our experiments to the same level of performance as if it could observe the whole map. We hypothesize that the slight difference in performance for the larger number of playouts is due to a "disconnection" between the real game state and the abstract game state, which manifests itself more prominently when we give the bot a larger computation budget. An interesting line of future work is to assess whether these results generalize to other bots as well.

Moreover, we would like to emphasize that our system did not lose any game, and the small percentage of games it did not win were ties, where our system could not locate the last remaining enemy units, due to the issue discussed in Section 3.3.6.

3.6.2 Conclusions

These experimental results indicate that the *Perfect Memory* believe state sampling strategy only decreases the performance of a game tree search algorithm (like MCTS) by 8%-15% compared to having access to the whole game state in μRTS, and achieves the same performance of having access to the whole game state in StarCraft. Therefore, despite the evidence shown in Section 2.6, *determinization* is able to handle RTS games with partially observable game states with a minor penalty in small domains (μRTS) and without any statistical significant penalty in large domains

(StarCraft). The "disconnection" problem between the real game state and the abstract game still needs to be addressed in order to have better performance, for instance, we would like to explore the possibility to have a mechanism to validate/ensure that a region with friendly and enemy units is fully observable or not (otherwise, while the bot believes that a region is fully observable, there might be parts of it that are not, where enemies might be located). Basically, we need to ensure that if the high-level MCTS search believes a region is observable then the bot can observe the whole region to prevent disconnections between the abstract representation and the actual game state.

Chapter 4: Spatial Reasoning

Spatial reasoning involves the location and movement of objects in a space [155]. For example, when a child rotates a triangular prism to fit the block into the hole, she is employing spatial reasoning. In the context of RTS games, an AI agent needs to reason about where to build the next base or how to navigate to avoid enemy fire. Therefore we need to understand the terrain and the surrounding objects that can affect it.

This chapter tackles some of spatial reasoning problems present in RTS games. More specifically, it presents (1) an algorithm to divide a map into regions of tactical importance, (2) an algorithm to close narrow passages with buildings (a.k.a. walling), and (4) a navigation algorithm that takes into account different unit and map features to simulate an attack-and-flee behavior.

4.1 Terrain Analysis

In RTS games, the map does not have a fixed configuration like in classic board games such as Chess or Go. Therefore it is a common practice to analyze the map in order to generate a qualitative spatial representation over which to perform reasoning [86]. This analysis can help us improve tasks such as pathfinding and decision making based on the properties of the terrain. The Brood War Terrain Analysis (BWTA) tool [47] is used by the majority of the participants in the StarCraft AI competitions to analyze the game map. However, this library has high computational demands, and often fails to detect all relevant areas of a map.

This section presents (1) a new algorithm to terrain analysis that, given an input map, generates a set of regions connected by chokepoints (or narrow passages) and (2) an interface for pathfinding queries. Compared to previous related work [47, 89], our approach is significantly more efficient in terms of execution time and detects chokepoints and corridors more accurately.

4.1.1 Brood War Terrain Analyzer 2 (BWTA2)

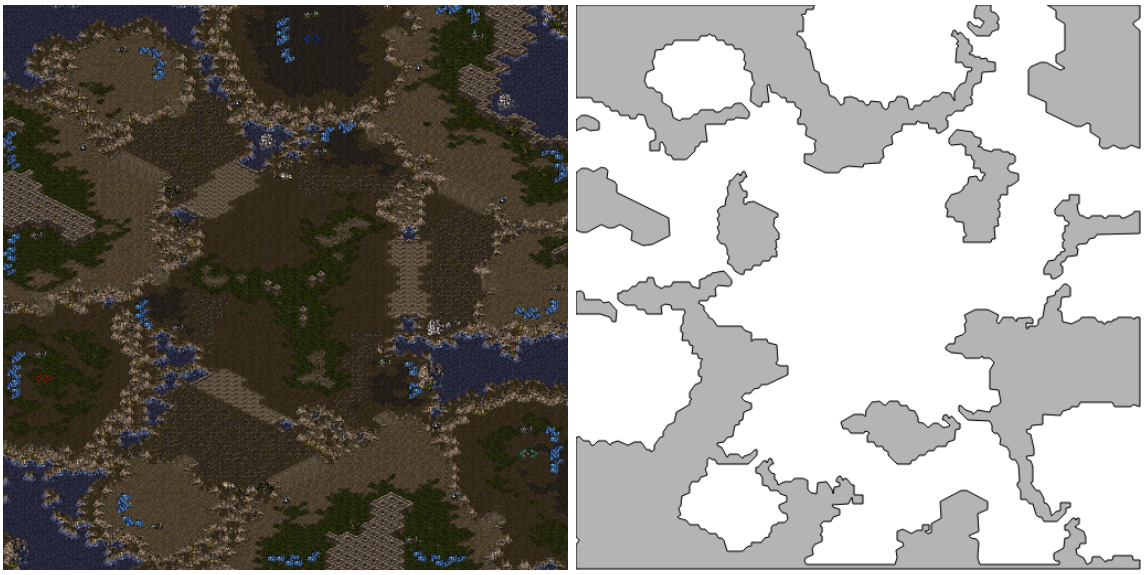
As mentioned, the majority of StarCraft bots use BWTA. Therefore, we decided to maintain as much as possible the same interface as BWTA to facilitate the migration; and we named our library *Brood War Terrain Analyzer 2* (BWTA2) to reflect the improvement over the previous well known library. Although the algorithm presented in this paper is different from BWTA, the global steps are similar to Perkins' algorithm:

1. Recognize Obstacle Polygons
2. Compute Voronoi Diagram
3. Prune Voronoi Diagram
4. Identify Nodes (Regions and Chokepoints)
5. Extract Region Polygons

For illustrative purposes, we show the result of each step for the StarCraft map Aztec, shown in Figure 4.1a.

Recognize Obstacle Polygons

The first goal is: given a raster image made with pixels (a map in our case), extract the polygons of the obstacles. This process called vectorization is a common step in the image processing community. First of all, we need to convert our image into a binary image to be able to detect the relevant shapes we are interested in. To do that, all pixels are marked as *white* if they are walkable by a ground unit, and *black* if they are not walkable. Moreover, there may be undestroyable objects such as neutral buildings that can make a pixel unwalkable. Now, we apply the component-labeling algorithm using contour tracing technique presented by Chang et al. [156]. This algorithm fulfills two purposes:



(a) Aztec StarCraft map. **(b)** Obstacle polygons detected in gray.

Figure 4.1: StarCraft map and obstacle polygon detected.

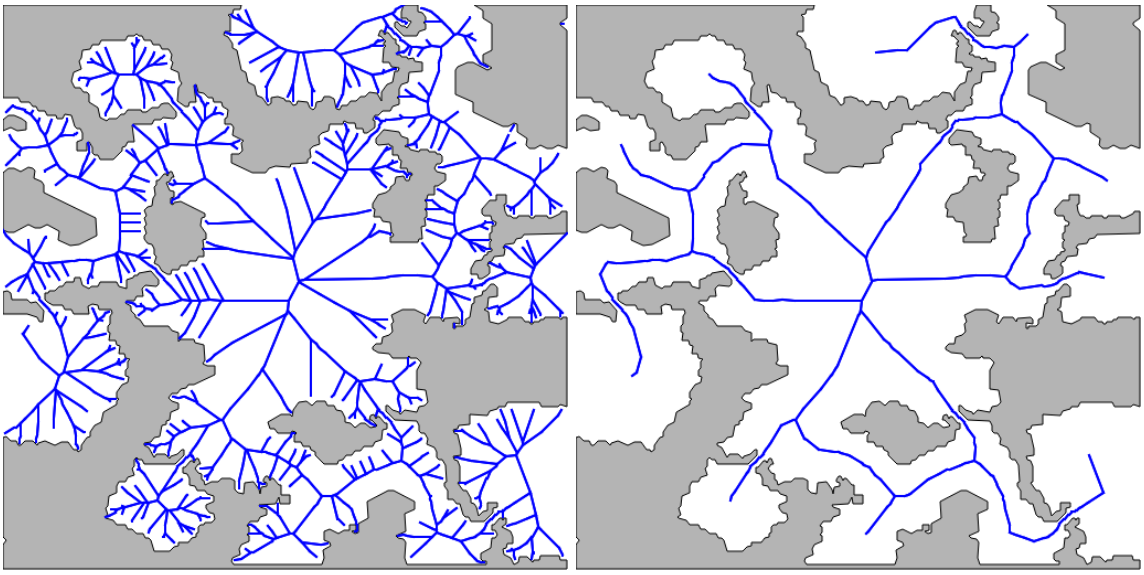
1. The contour tracing technique detects the external contour and possible internal contours of each component, i.e., it detects the outer ring and inner rings (or holes) of each obstacle polygon.

2. Each interior pixel of a component is labeled. This will make any query to get what polygon a pixel belongs to a $O(1)$ operation.

Now we have the contour of the obstacles as a sequence of pixels. In order to reduce the number of points that make up the contour we use the Douglas-Peucker simplification [157], and after that we iterate over all simplified points and if the distance to the border is less than a threshold (2 pixels in our experiments) we move the point to the border, we call this "anchoring to the border". This last step is to ensure that all the obstacle polygons remain attached to the borders after the simplification. This polygon simplification will help us to reduce the computation complexity of the Voronoi diagram. Figure 4.1b shows the obstacles detected for the Aztec map.

Compute Voronoi Diagram

In order to improve BWTA, BWTA2 has as secondary goal to reduce the library dependencies; therefore we tried to avoid to use the Computational Geometry Algorithms Library (CGAL) to compute the Voronoi diagram. Instead, we use the implementation in the Boost library (CGAL depends on Boost and other libraries to work). Boost uses Fortune's sweep-line algorithm [158] to compute the Voronoi diagram, while CGAL uses an incremental algorithm [159] that allows to add segments that may intersect at their interior (with a performance penalty). Since we know all the segments beforehand and all the polygons are ensured to be simple thanks to the polygon simplification step, we can use the sweep-line algorithm that significantly outperforms the incremental one used by BWTA. In order to avoid infinite Voronoi segments, we also add the necessary segments to close the border of the map (not drawn in our Figures). Boost returns the Voronoi diagram as a half-edge data structure. For our purposes we simplify this structure to a regular graph represented by an adjacency list. We only add to the graph the Voronoi points that do not lie inside of an obstacle polygon. Figure 4.2a shows our computed Voronoi diagram.



(a) Voronoi diagram of walkable areas. **(b)** Medial axis from pruning the Voronoi diagram.

Figure 4.2: Voronoi diagram and pruning.

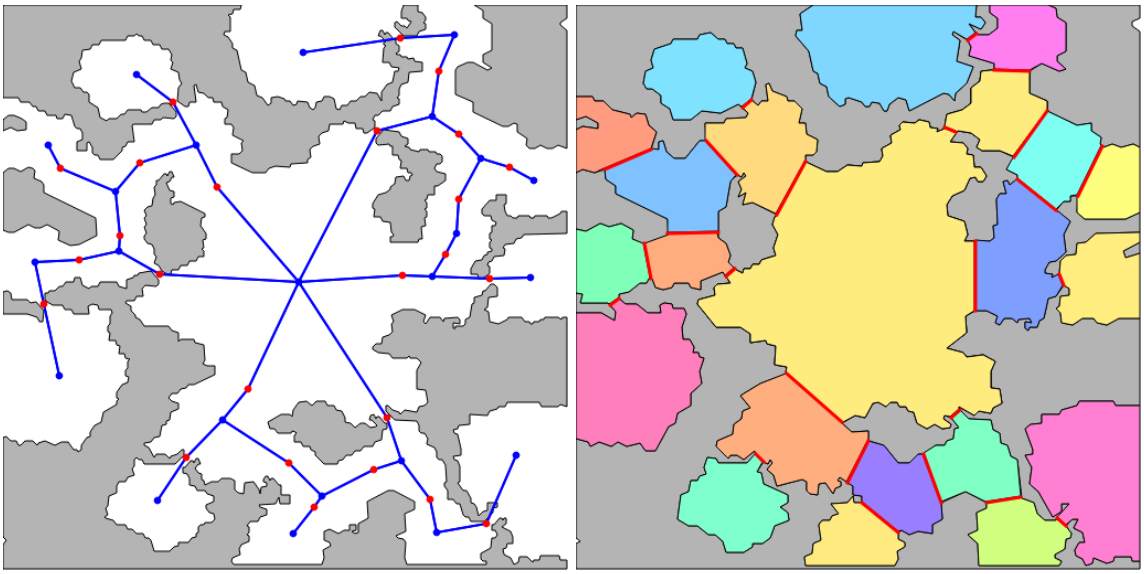
Prune Voronoi Diagram

In this step we want to compute the *medial axis* given the Voronoi diagram by removing the "hairs" of the Voronoi diagram. To do this, we compute the distance to the closest obstacle for each vertex in our graph. These queries can be optimized using a R-tree structure [160] with a packing algorithm [161]. Adding all the segments to the R-tree will let us perform a query to know the closest obstacle of a point, let us call this the *radius* of the vertex. Now, we use all the leaf vertices in the graph to initialize the list of candidate vertices to be pruned. Each vertex in the list is evaluated once, if the vertex is too close to an obstacle (has a radius less than 5 pixels) or if the parent is farther to an obstacle, we remove the vertex and the edge. After removing the edge, if the parent vertex becomes a leaf, we add it to the list of candidate vertices to be pruned. Figure 4.2b shows how the resulting medial axis still captures the structure of the map but with less complexity.

Identify Nodes (Regions and Chokepoints)

Now we identify all local maxima, i.e., vertices whose radius is bigger than any of their neighbors. Notice that this corresponds to the biggest inscribed circle of a region; and all local minima, i.e., vertices whose radius is smaller than its neighbors, are chokepoints of the map or narrow passages. Our proposed algorithm only iterates once over each vertex, and therefore it has a $O(|V|)$ time complexity. Starting from a leaf vertex, we mark that vertex as a region node (after the pruning, all remaining leaf vertices are local maxima) and we add its children to the list of vertices to explore. Then for each vertex in the list we proceed as follows:

1. We add all unvisited children to the list of vertices to explore.
2. If the vertex has a degree other than two, we mark it as a *region node* since it is a leaf or an intersection point.
3. When a vertex has two children and it is a local minima, if the parent is also a local minima we mark as a *chokepoint node* only the vertex with the smallest radius, otherwise we mark the current vertex as a *chokepoint node*.
4. In the other hand, when a vertex with two children is a local maxima, if the parent is a local



(a) Final nodes detected (chokepoints in red and center of regions in blue).

(b) Map decomposed in regions.

Figure 4.3: Nodes detected and map decomposed in regions.

maxima we mark as a *region node* the vertex with the biggest radius, otherwise the current vertex is marked as a *region node*.

The next step is to simplify our graph, first we create a new graph with only the marked nodes (region or chokepoint) and their connections; and then since all paths must alternate between a chokepoint and a region node, all connected regions (those that are intersections with another region) are merged, keeping only the region with the biggest radius. Our presented algorithm is more efficient than the one presented by Perkins since we only visit the vertices $|V| + |V| + |N|$ (the first time to mark the nodes, the second time to simplify the graph to N nodes, and the third time to merge consecutive regions). Figure 4.3a shows the region nodes (blue dots) and chokepoints nodes (red dots) identified by our algorithm.

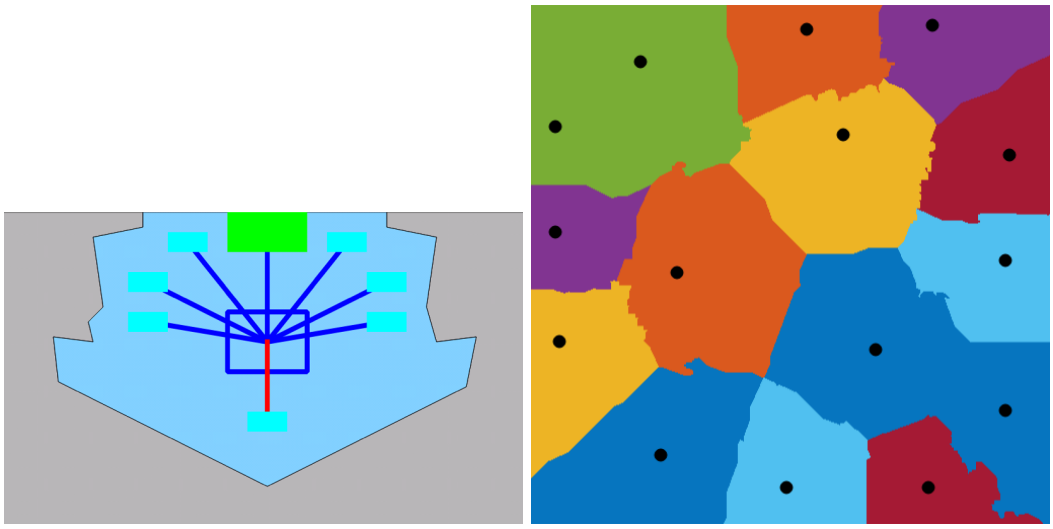
Extract Region Polygons

Once the chokepoints nodes have been identified, we use the previous R-tree structure to do an iterative query to find the two obstacle points that crosses each chokepoint node. Now, to generate the region polygons we start with a square covering the whole map and we compute the difference with the obstacle polygons to generate the walkable polygons. Finally, we compute the difference of walkable polygons and the chokepoints segments to split them into the region polygons. Figure 4.3b shows the final region division. As a final step, we generate our internal data structure where: 1) each *Region* has its own polygon, the center point from the region node (notice that this position is ensured to fall inside the polygon in opposition to a centroid of a polygon), and a set of adjacent *Chokepoints*; and 2) each *Chokepoint* has a segment, a middle point and two connected *Regions*.

Extra cache operations for AI queries

BWTA2 performs several extra computations to speed up common operations that most AI agents use.

- A component-labeling algorithm is performed to the walkable pixels in order to know what regions are connected and which ones are islands.



(a) Example of resources assigned to a base location.

(b) Multi-seed flood-fill distance to closest base location.

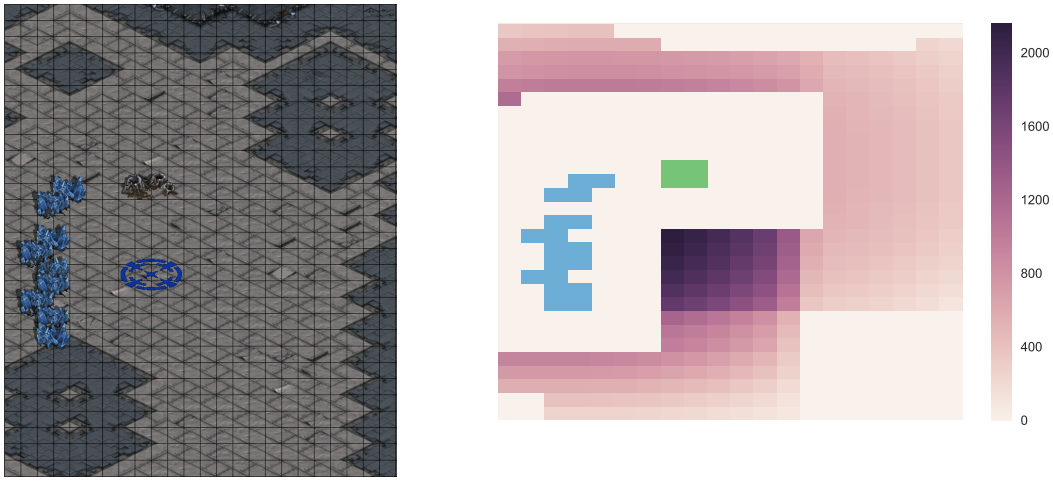
Figure 4.4: Base location and its resources and the closest base location map.

- We identify all potential locations to build a base. To optimize resource gathering, bases are located equidistant to a cluster of resources. We identify cluster of resources using a DBSCAN algorithm [162] with the following constraints: the minimum distance to a group resources is 6 build-tiles, the minimum resource cluster size to be considered is 3. After that, for each cluster we build an influence map around each resource based on distance (Vespene geysers have 3 times more influence to compensate that there is usually only one) and the best score in the cluster bounding box is marked has the best build location (Figure 4.5). Figure 4.4.a shows an example of resources associated with a base location (blue lines), and the special case where a small mineral (a resource with less than 10 minerals) is blocking the best base location (red line).

- Despite decades of research, pathfinding is still an expensive computation and an open research area. BWTA2 implements HPA* [79] which uses the concept of *rooms* and *gates* as a higher-level structure to perform the search. Here, we use our division of *regions* and *chokepoints* as an abstraction for our search. The idea of using smart terrain decomposition for pathfinding was also explored by Halldórsson et al. [89]. One of the major speedups of the algorithm is achieved by caching the distance between all *gates* (or *chokepoints* in our case).

- Closest point of interest. A common query is the distance to a closest object (base location, chokepoint, ...). A *multi-seed flood-fill* algorithm is performed for each relevant object. The idea of a *multi-seed flood-fill* algorithm is to initialize a FIFO list with the different seed positions and use this list to pop the next position to check and push the following positions to explore; we also customized it to increase the cost once a position switches from walkable to unwalkable for the first time. Hence, the result shows the closest object for a ground unit. Figure 4.4.b shows the closest distance to a base location.

- Region visibility coverage. The regions generated are usually bigger than the sight range of the units. To let an AI ensure that at any given moment the whole region is visible (not under the fog-of-war), we computed the minimum set of locations where an AI would have to place units in order to have the whole region in sight range. The locations are computed using a spiral search from the center of the region and adding locations every time we explore a place that is farther than the minimum sigh range of all the marked coverage locations. Figure 4.6 shows



(a) Cluster of resources in Benzene map.

(b) Influence map of resources (blue are minerals, green is gas).

Figure 4.5: Best base location detection.

Table 4.1: Time in seconds to analyze each map by BWTA and BWTA2.

Map Name	Map Size	BWTA	BWTA2
Destination	128x96	54.15	1.35
Heartbreak Ridge	128x96	53.55	1.31
Benzene	128x112	55.82	1.40
Aztec	128x128	66.83	1.60
Andromeda	128x128	78.16	1.77
Fortress	128x128	49.19	1.60
Python	128x128	43.70	1.64

an example of the coverage points (marked in blue) found from each region center (marked in red) for Benzene map.

4.1.2 Experimental Evaluation

We compare the algorithm presented in this paper (BWTA2) against BWTA, since this is the only open source state-of-the-art algorithm. We analyzed seven different popular maps used in the StarCraft AIIDE competition that cover different map size ranges: Destination, Heartbreak Ridge, Benzene, Aztec, Andromeda, Fortress and Python. We compare the time to analyze the map by each algorithm. Table 4.1 shows that our proposed algorithm is more than 26 times faster than BWTA, and it only takes less than 2 seconds to analyze complex maps. See Appendix C for the result of StarCraft AIIDE competition maps analyzed by BWTA2.

On the other hand our approach does a better job finding chokepoints, especially long corridors or small symmetric regions. Figure 4.7 (left) shows the chokepoints detected by BWTA2 in red (regions in blue are not merged); Figure 4.7 (right) shows the chokepoints detected by BWTA in red and the missing chokepoints that BWTA2 was able to find in green. As mentioned, the missing points are long thin corridors or small regions with two chokepoints. Since regions are symmetrical, we can observe how sometimes BWTA detects the same chokepoints in the symmetric region and other times it only detects one of the chokepoints.

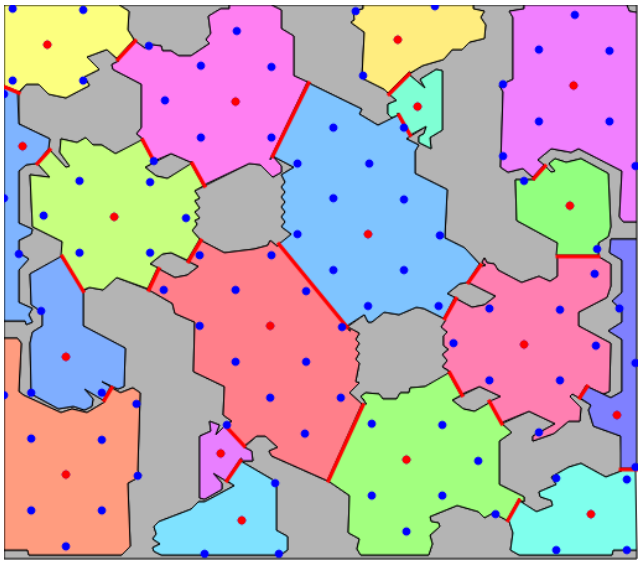


Figure 4.6: Coverage points (in blue) to ensure the visibility of each region in Benzene map.

Table 4.2: Time in milliseconds to compute the distance between bases in Aztec map.

Tile Positions	A*	HPA*
(9,84) and (69,7)	41.13	0.95
(9,84) and (118,101)	46.72	0.73
(69,7) and (118,101)	41.38	1.09

4.1.3 Applications to Pathfinding

The most common use of terrain analysis is for pathfinding or navigation. As we mentioned there are several algorithms capable of using the information of terrain analysis to guide the pathfinding search. BWTA2 implements the idea of HPA* [79] to use the *chokepoints* as *gates*. Besides the algorithm being near-optimal, in RTS games we can have hundred of units computing distance checks, usually for planning a long-term plan, therefore a fast approximation like HPA* is perfect for our case. Keep in mind that the environment in RTS games is dynamic (i.e., changes can occur while units are moving), the navigation system should be combined with a reactive one capable of handling the frame-by-frame unit control. Here we are only considering path planning. Table 4.2 shows a time comparison in ms between A* and HPA* implemented in BWTA2, showing that HPA* can be 64 times faster than A*; a significant difference, especially important in games with a lot of units moving in real-time.

4.1.4 Applications to Strategic Decision Making

Lidén [163] showed how terrain analysis can help other tasks beyond pathfinding. Chokepoints or waypoints can be used as key points to calculate visibility to perform an intelligent attack positioning such as flanking or squad coordination.

Additionally, we presented how to detect potential base locations (Subsection 4.1.1) to decide the next base expansion. Quantifying the tactical importance of each region can help establish the most profitable location from the point of view of map control, or which the best enemy base to attack. Finding the closest relevant object is an expensive query executed several times during a game. We cached these queries to improve the overall AI agents' performance.

Finally, another application of terrain analysis is game tree search. For example, Uriarte and Ontañón [164] used a map decomposition as a way to define an abstraction over which to employ

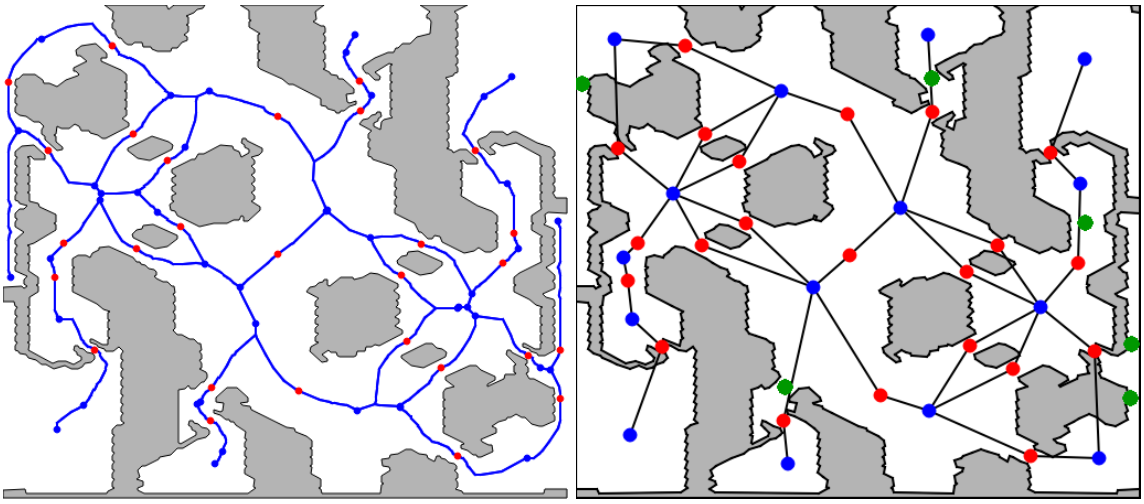


Figure 4.7: Chokepoints in red detected by BWTA2 (left) and missing in green by BWTA (right).

Monte Carlo Tree Search.

4.2 Walling

A classic tactic to defend a base in RTS games is to make a wall, that is, to construct buildings side by side in order to close or to narrow the base entrance. Closing a base gives the player extra time to prepare a defense, or helps him to hide some pieces of information about his current strategy. Narrowing an entrance creates a bottleneck, which is easier to defend in case of invasion. We focus only in walls constructed by buildings, discarding small narrow passages that can be closed by small units like workers.

In StarCraft, the map space is defined by two grids: The *walk grid*, where each cell is an 8×8 pixels square, and the *build grid*, where each cell is a 4×4 walk tile square (*i.e.*, of 32×32 square of pixels). Each cell in the build grid is called a *build tile*. Moreover some build tiles are *buildable* and some are *not-buildable*. The approach presented is applicable for any RTS game for which such a *build grid* exists.

We define two properties of buildings: their *build size*, and their *real size*. The build size is a pair (w, h) of build tiles. In order to create such a building, we need a rectangle of buildable tiles in the map (w build tiles in width and h build tiles in height). The real size is a pair (w_p, h_p), such that $w_p \leq 32 \times w$ and $h_p \leq 32 \times h$, representing the actual size of the building in pixels once it's constructed in the game, where 32 is the size in pixels of a build tile in StarCraft (but might be different for other RTS games). The real size of a building can then be smaller than its build size. This is actually always the case in StarCraft and it means that two buildings constructed side by side are still separated by a gap which may be big enough to let small units enter, like Zerglings, Marines or Zealots in StarCraft.

The walling problem is an optimization problem described as follows: Given a chokepoint and two buildable tiles s and t (start and target tiles), choose a set of buildings B and place them on the map such that:

- All buildings of the set are part of a wall (i.e., they are contiguous), and the tiles s and t are covered by buildings.
- A target optimization function f about the wall may be minimized (such as number of buildings or number of gaps).

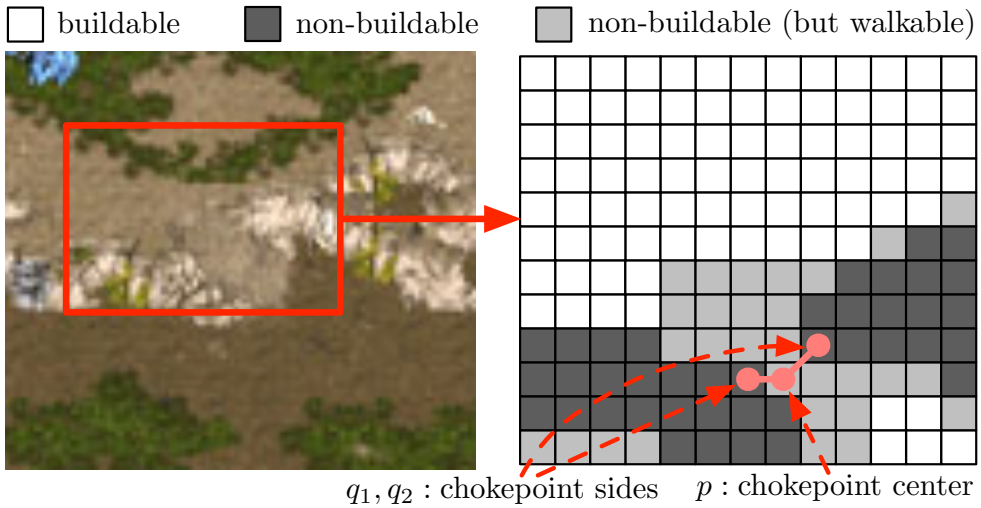


Figure 4.8: Mapping of a portion of a StarCraft map to a grid of build tiles.

4.2.1 Walling via Constraint Optimization

The approach to walling presented in this paper consists of three main stages: *chokepoint identification*, *wall specification*, and *wall creation*, which correspond, respectively to: identifying a place in the map where the wall can be created, determining the exact coordinates where we want to generate the wall, and finally determining which exact buildings do we need and in which coordinates. The following subsections describe each of these stages in turn.

Chokepoint Identification

For the chokepoint identification we use the map decomposition implemented in the BWTA2[1] (and explained in Section 4.1). This algorithm decomposes a map into regions and chokepoints that connect exactly two regions. It first computes *obstacle polygons* (representing each of the non-walkable regions). Then, using the edges of these polygons, a Voronoi diagram of the line segments is generated. Then after pruning some vertices of the resultant Voronoi diagram, the algorithm looks for the vertices with degree two and with a small distance to the nearest obstacle polygon. Those vertices are marked as chokepoints. Once we have all the chokepoints, we filter the chokepoints that are too small (smaller than 3 build tiles) or too big (larger than 12 build tiles) to build a wall or the ones in which building a wall does not make sense (where there are no buildable cells around the chokepoint).

Wall Specification

In the second stage, given a chokepoint $\langle p, q_1, q_2 \rangle$, where p are the coordinates of the chokepoint, and q_1 and q_2 are the coordinates of the sides of the chokepoint (i.e., the coordinates of the two walls in each side of the chokepoint, as illustrated in Figure 4.8), we want to determine the specific points s and t where the wall should start and end. We distinguish three main situations:

- **Buildable Chokepoint:** When all the cells in the grid in the segments that connect q_1 to p and p to q_2 (except for q_1 and q_2) are buildable. This is the simplest scenario, and s and t can be determined using the following procedure (illustrated in Figure 4.9):

  1. Using Bresenham's line drawing algorithm [165], we trace a line from q_1 to p. The first point in this line that is buildable is labeled as s.

[1] BWTA2 can analyze maps off-line, and can be downloaded from: `https://bitbucket.org/auriarte/bwta2`

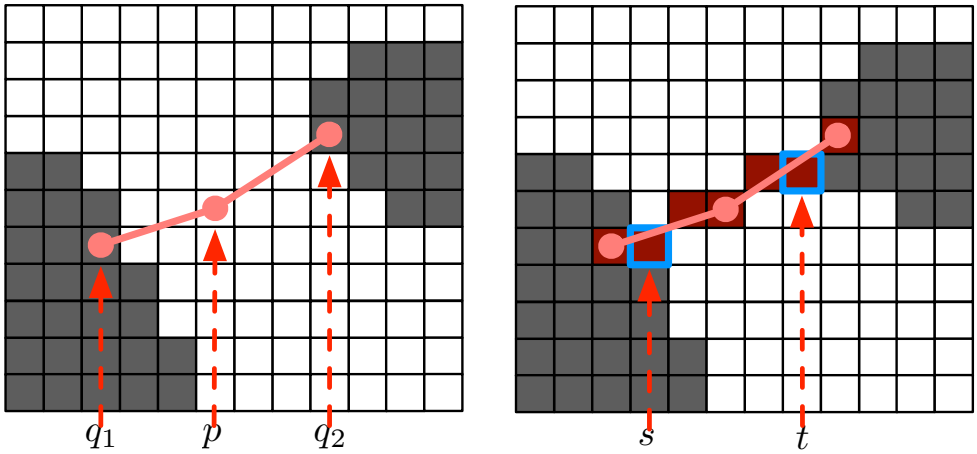


Figure 4.9: Determining the start (s) and target (t) coordinates of a wall when the chokepoint is buildable.

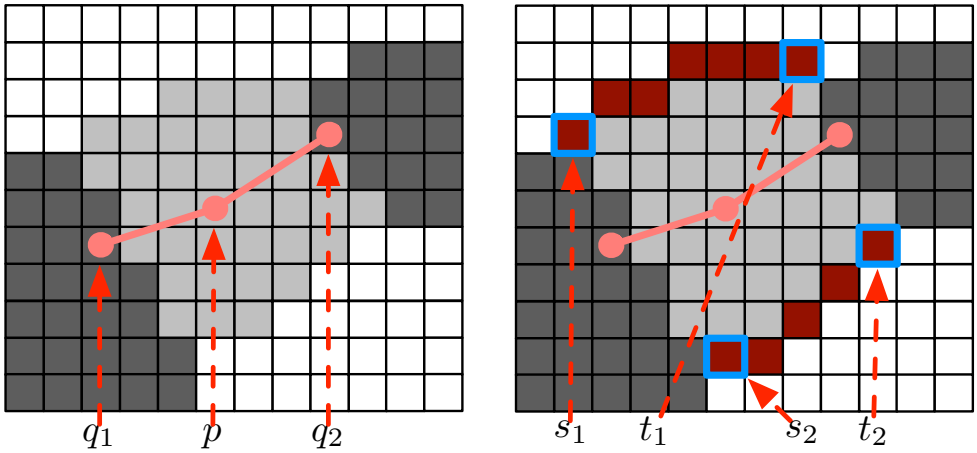


Figure 4.10: Determining the start and target coordinates of a wall when the chokepoint is non-buildable. There are two possible walls in this case s_1 to t_1 and s_2 to t_2.

2. Using Bresenham's line drawing algorithm [165], we trace a line from q_2 to p. The first point in this line that is buildable is labeled as t.

- **Non-Buildable Chokepoint:** When all the cells in the grid in the segments that connect q_1 to p and p to q_2 (except for q_1 and q_2) are not-buildable. In this case, the choke point is walkable, but not-buildable. Therefore, the wall cannot be built in the chokepoint itself. s and t can be determined as follows (Figure 4.10):

  1. We run a *flood-fill* algorithm starting from point p to determine the set of connected (using 4-connectivity) cells that are not-buildable, but walkable (e.g., the set of gray coordinates in Figure 4.10). Let us call this set R.
  2. We find the set of buildable cells in the map that have at least one neighbor (4-connectivity) in R and at least one neighbor outside of R (the set of dark red cells in Figure 4.10). Let us call this set B (border).
  3. We split the set B into disjoint sets $B_1, ..., B_n$, such that all the cells in B_i are neighbors (8-connectivity) and no cell in B_i is a neighbor of any cell in B_j if $i \neq j$ (there are two such sets of cells in Figure 4.10).

4. For each set B_i, determine the two extremes s_i and t_i, such that s_i and t_i are the pair of points from B_i that are further from each other that satisfy that they both have at least one neighbor (8-connectivity) that is not-buildable and not-walkable. Each of these pairs s_i, t_i determines the start and target points of a possible wall (two such walls are found in Figure 4.10).

Notice that it is theoretically possible to have $n > 2$, but in this case more than one wall would be potentially needed to block the chokepoint. In this situation, we just consider the chokepoint as not a candidate for blocking with a wall.

- **Mixed Chokepoint:** Although not common in StarCraft maps, chokepoints with a mix of buildable and non-buildable cells in the segments that connect q_1 to p and p to q_2 sometimes occur. However, we do not consider this scenario in our approach.

Given the start and target coordinates of a wall: s, t, we determine whether the wall is feasible by computing the distance between s and t. If s and t are too far apart, constructing the wall might not bring any significant advantage, and thus it should not be attempted. On the other hand, if s and t are too close, there might not be space for placing buildings, and the wall might not be possible. For the experiments presented in this paper, we only considered those situations where the following conditions where satisfied: $4 < |t_x - s_x| \leq 12$ and $3 < |t_y - s_y| \leq 9$.

Wall Creation

We express walling as a Constraint Optimization Problem (COP). A COP is a tuple (V, D, C, f) where V is a set of k variables, D the set of domains of each variable, *i.e.*, sets of values that variables can take, C is the set of constraints upon V and $f\colon V^k \to \mathbb{R}$ is a k-ary objective function to minimize or maximize. A *configuration* is a mapping of each variable in V to a value in their domain in D. A configuration may then satisfy none, some or all constraints of the COP instance. In this latter case, the configuration is called a *solution*.

In [92], a COP is presented to model the walling problem, where V is the set of buildings that may compose the wall, D the set of possible positions for each building of V. Two objective functions f_v and f_h are present to respectively minimize the vertical size and the horizontal size of the largest gaps between buildings. This leads to a complex multi-objective optimization problem, where goals do not always prevent foes to go through the wall, but where an imperfect wall is considered to be better if its largest gap can let pass only one zergling rather than two side by side.

We propose a different approach: V and D remain respectively the set of buildings we can use to build the wall and the set of possible positions for these variables, but we use the following set of constraints C (illustrated in Figure 4.11):

- **Overlap**: buildings do not overlap each others.
- **Buildable**: buildings can be built on their positions (do not overlap unbuildable tiles).
- **NoHoles**: no holes of the size of a build tile (or greater) in the wall.
- **StartingTargetTile**: there are exactly one building constructed on a given starting tile s, and one building (it can be the same one) on a given target tile t.

In fact, Overlap, Buildable and NoHoles are sufficient to make a wall. We added the StartingTargetTile constraint to help the solver to find how to surround a chokepoint. In the other hand, some RTS games might require additional constraints. For example, if using the Protoss race in StarCraft, one should add a fifth constraint to consider *Pylons*.

There are few works in RTS game AI using constraint programming techniques, and even fewer using meta-heuristics. Among others, branch and bound algorithms (not a meta-heuristic) have been used to optimize build order [28]. Genetic algorithms have been used off-line to optimize build order, but with multiple objectives, analyzed in [166] and a population-based algorithm has been used for multi-objective procedural aesthetic map generation [167]. To solve our COP instance, again we

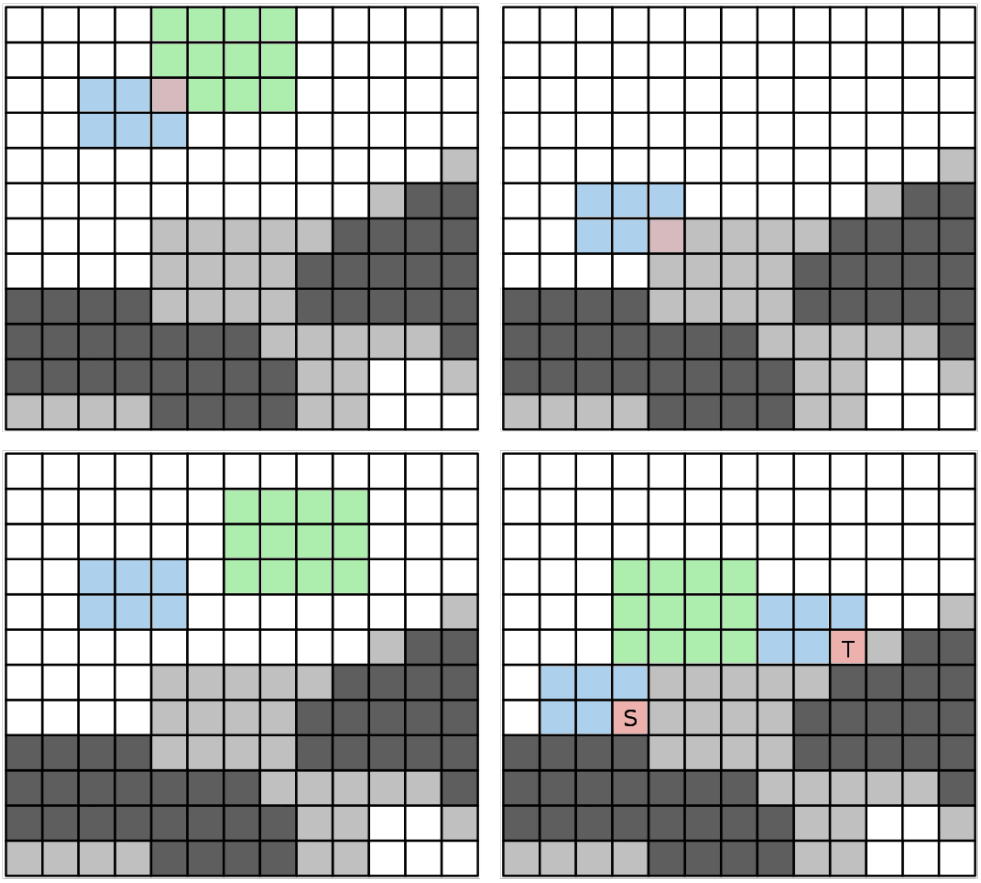


Figure 4.11: Constraint are: Overlap (upper-left), Buildable (upper-right), NoHoles (bottom-left) and StartingTargetTile (bottom-right).

have chosen a different technique than in [92]. In that previous work, an ASP logic program has been written and is solved by the ASP solver Clingo. Even if it allows finding an optimal solution, time computations may not fit real-time games like RTS, since their formulation requires up to 200 ms per ASP solver call. In contrast, in the rules of the annual AIIDE StarCraft competition, it is clearly specified that a bot where the time computation during a frame exceeding 55 ms more than 200 times loses automatically the game. In order to have faster optimization, we opted for a local search method based on the Adaptive Search algorithm [168, 169], which is up to our knowledge one of the fastest meta-heuristic to solve constraint-based problems. Even if walling might only be used a few times during a game, aiming at fast computations is important since our approach can be used for many other tasks in RTS games, such as base layout. Thus, one can run our solver to make a wall at a base entrance, but also to manage building placements into the base.

The main idea of the Adaptive Search algorithm is the following one: a cost function is declared for each kind of constraint in the COP telling how much a constraint is far to be solved within the current configuration. The output of such a cost function is a *constraint cost*. If the cost of a constraint c is zero, it means c is currently satisfied. One can then give a global cost to a configuration, usually by adding the cost of each constraint in the COP instance. The originality of Adaptive Search is that it projects constraints cost on variables, *i.e.*, it sums the cost of all constraints where a given variable occurs. For instance, if a variable x appears in constraints c_1 and c_2 only, then the projected cost on x will be the cost of c_1 plus the cost of c_2. This allow the algorithm to know what variables are the most responsible for the violation of constraints, and then permit to apply a sharper variable selection heuristic.

Table 4.3: Overall Experimental Results over 48 different problems, extracted from 7 different maps from the StarCraft AI competition. Results are the average of 100 runs in each problem (total of 4800 runs per configuration).

#Attempts	1	2	3	4	5	10	20	50
Average Cost	1.59	0.58	0.33	0.18	0.13	0.03	0.01	0.0006
Solved %	45.83%	71.10%	80.70%	88.45%	91.58%	97.23%	99.18%	99.95%

The main problem with meta-heuristics is that these methods can be trapped into a local minimum, *i.e.*, a non-optimal configuration where there are any local moves leading to a better configuration. To escape from these situations, one classical and efficient method is to simply restart the algorithm from a random-selected configuration. Thus, our COP solver (called GHOST [94]) has two different behaviors:

Satisfaction: The user only asks for a wall, in this case the solver tries to find a wall satisfying our four constraints within one run limited by a timeout (for instance 20 ms).

Optimization: The user only asks for an optimized wall, for instance trying to have as few large gaps as possible. Then, the solver will launch a series of runs limited by the given timeout (like 20 ms), and saves what is the best solution found so far, according to the objective function. It stops when: 1) It finds a perfect solution *i.e.*, an optimization cost equal to zero (which is not possible for some objective functions like minimizing the number of buildings); or 2) It reaches a global timeout, for example 150 ms. Launching a series of small runs is important because it means we can slice a 150 ms optimization run into several 20 ms independent pieces that can be executed in different frames; so that an 150 ms optimization run do not exceed the 55 ms limit per frame during competitions.

The walling problem offers many interesting optimization opportunities. We propose the following objective functions to minimize:

- **Building**: the number of buildings in the wall,
- **Gap**: the number of significant gaps in the wall,
- **TechTree**: the required technology level in the game. In games like StarCraft, some buildings are unlocked after developing specific technologies. Therefore, it is interesting to build walls with buildings of low technology.

We propose to use a single function f to minimize chosen among the three presented above. The most complex and interesting one is the objective function trying to reduce the number of gaps in the wall that are big enough to let small units to pass. This function can be adapted according to the opponent race. Indeed, the smallest units in the game are Zerg's Zergling (16 pixels wide $\times$16 pixels high), Terran's Ghost (15×22) and Terran's Marine and Medic (17×20), and Protoss' Zealot (23×19), if we don't consider Protoss' Scarab (Reaver's projectiles, 5×5 pixels). We report experiments assuming a Zerg opponent.

4.2.2 Experimental Evaluation

This section presents an empirical evaluation of our approach in the RTS game StarCraft. We only considered Terran buildings without their extensions. The main goals of our evaluation are: 1) to determine the quality of solutions that can be achieved under the tight timing constraints in RTS games, 2) to determine how much can optimization improve the base solutions provided by a satisfaction run.

In order to evaluate our approach we used seven maps from the StarCraft AI competition: *Benzene*, *Aztec*, *Circuit Breaker*, *Python*, *Heartbreak Ridge*, *Andromeda*, and *Fortress*. From each of

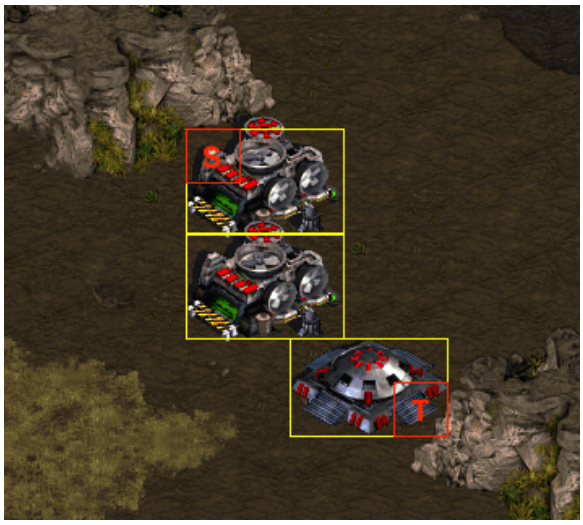


Figure 4.12: Example of wall solved from the start (s) to target (t) coordinates.

Table 4.4: Results over 48 chokepoints extracted from 7 StarCraft maps. Results are the average of 100 runs for each chokepoint. Each calls of GHOST lasts for 150 ms

Objective	Satisfaction Run	Optimization Run	% Solved (Opti)
Building	4.05	2.56	98.04%
Gap	1.32	0.03	97.50%
TechTree	1.99	1.35	97.54%

those maps, we extracted a collection of chokepoints and employed the methods described in this paper to determine the start and target coordinates of the walls. In total this resulted in 48 wall specifications.

Table 4.3 shows the results obtained using satisfaction runs in the 48 chokepoints using a timeout of 20 ms. Specifically, we show both the average solution cost (0 means that a wall was found) and the percentage of times that a wall was found while building walls for the 48 chokepoints. The *Average Cost* indicates how far we are from a solution. Thus, the evolution of this value in Table 4.3 is more meaningful than the values themselves. The presented results are the average of 100 runs. Each column in Table 4.3 shows the results when we allow our system to restart a different number of times. The first column shows results when the solver only has one attempt (with a timeout of 20 ms), solving 45.83% of the problems. The second column shows results when our solver has two attempts (each time with a timeout of 20 ms), and showing that 71.1% of the times at least one of the attempts resulted in a solution. As Table 4.3 shows, when giving our solver at least 5 attempts, more than 90% of the problems were solved. This is remarkable, since, due to the small timeout used, it means that our solver can run once per game frame. Thus, after 5 game frames (*i.e.* 0.21 seconds of gameplay) the probability of having found a solution to a walling problem is high. With a higher number of attempts the probability of success stabilizes at 99.95%. Figure 4.12 shows an example of a wall proposed by our solver from the start (s) to target (t) coordinates.

Satisfaction runs in Table 4.4 are GHOST runs without any objective functions and with a satisfaction timeout of 160 ms. We measure their average number of buildings, significant gaps and technology level in order to match them with optimization runs. Optimization runs are slightly disfavored since their global timeout is 150 ms, thus 10 ms shorter than satisfaction runs. For optimization runs, GHOST launched a series of successive runs of 20 ms, with a global timeout of 150 ms. We can see in Table 4.4 that optimization runs lead to real improvements compared to satisfaction runs. This is particularly true with the Gap objective, where significant gaps are almost

Table 4.5: Percentage of solutions found for each map

Map Name	Solved
Python	100
Heartbreak Ridge	100
Circuit Breaker	99
Benzene	99
Aztec	97
Andromeda	96
Fortress	90

Table 4.6: Average time over 20 runs to find a solution

Chokepoint width (pixels)	Buildings needed	GHOST Avg. time (ms)	Clingo Avg. time (ms)
65	2	46.8	362.8
250	2	33.5	408.8

completely eliminated: 4,680 of 4,800 walls have been found (97.50%), and 4,527 of them are perfect walls (*i.e.*, without any significant gaps). In other words, 96.73% of walls found by GHOST are perfect.

Table 4.5 shows the percentage of walls found in each of the seven maps from where chokepoints were extracted. These numbers correspond to pure satisfaction runs, since they do not differ significantly from optimization runs. We can see that GHOST has more difficulties to find a wall for Fortress chokepoints. Actually, it is failing from time to time on the same chokepoint, where a valid solution can only be achieved by using two 3×2-sized buildings.

We also performed a comparison between GHOST and the state-of-the-art for the walling problem, in this case the work presented by Certicky [92]. First, we limited the number of possible buildings to be considered to the number of buildings in Certicky's work (2 Barracks and 4 Supply Depots) and recorded the time to find a solution in two different cases: a small chokepoint (width of 65 pixels) and a big chokepoint (width of 250). In Table 4.6,[2] we can see the average results of both solvers. The times include the computation needed for setting the solvers parameters, the time to solve the problem, and the time to parse the solution (in the case of Certicky's solution, we need to make an external call to Clingo solver). As we can see, GHOST is at least 7.8 times faster than Clingo in our experiments. GHOST is faster in the wider chokepoint because the smaller one is a ramp and there is an overhead of computing the extremes of the ramp.

4.3 Kiting

In this Section we tackle the problem of how can a friendly unit (or group) attack and destroy an enemy unit (or group) while minimizing the losses amongst the friendly units. Specifically, we face the problem using a **kiting behavior**. Intuitively, the basic idea of kiting is to approach the target unit to attack, and then immediately flee out of the attacking range of the target unit. More specifically, given two units u_1 and u_2, unit u_1 exhibits a **kiting** behavior when it keeps a safe distance from u_2 to reduce the damage taken from attacks of u_2 while the target u_2 keeps pursuing u_1.

Furthermore, we say that a unit u_1 performs **perfect kiting** when it is able to inflict damage to u_2 without suffering any damage in return, and we say that u_1 performs **sustained kiting** when it is not able to cause enough damage to kill unit u_2, but u_2 is also unable to kill u_1.

[2]Comparison experiments can be found at `https://bitbucket.org/auriarte/bwta2/src`

Table 4.7: Vulture and Zealots attributes (times are measured in game frames, and distances in pixels)

Unit	HP	Speed	Deceleration	Accel.	Turn	AttackTime	AttackRange
Zealot	160	4.0	0	2	1	2	15
Vulture	80	6.4	0	9	3	1	160

Sustained kiting is useful in many situations. For example, in many RTS games players need to send units to "scout" what the opponent is doing; these scout units can exploit sustained kiting behavior while scouting the enemy base to remain alive as much as possible, and to make the enemy waste as much time as possible trying to destroy the scout. However, the approach presented in this paper aims at perfect kiting. The execution of perfect kiting depends on the characteristics and abilities of the unit and the target unit. In this paper we will present a method to detect when perfect kiting is possible, and provide an algorithm to execute this behavior successfully.

4.3.1 An Influence Map Approach to Kiting

Our approach to model kiting behavior divides the problem in three subproblems: 1) deciding when kiting can be performed, 2) creating an influence map to guide the movement, and 3) target selection. The following three subsections deal with each of those subproblems in turn.

When Can Kiting Be Performed?

It is not always possible to successfully perform a kiting behavior. Some conditions must be met in order to execute it. This section focuses on identifying such conditions. Let us start by introducing some preliminary concepts.

Each unit in a RTS game is defined by a series of attributes, such as movement speed or attack range. In particular, we are concerned with the following attributes:

- **Speed:** top speed of the unit.
- **Acceleration:** time that a unit needs to get to top speed.
- **Deceleration:** time that a unit needs to stop.
- **Turn:** time that a unit needs to turn 180 degrees.
- **AttackTime:** time that the unit needs to execute an attack (notice this is different from the "attackCoolDown" attribute that some RTS games have, which defines the minimum time between attacks).
- **AttackRange:** maximum distance at which a unit can attack.
- **HP:** hit points of the unit. When the HP are equal or lower than 0 the unit is removed from the game.
- **DPS:** Damage per second that the unit can inflict to a target.

Given two units u_1 and u_2 defined by the previous attributes, u_1 can kite u_2 only when two conditions are met: 1) when u_1 is faster than u_2, and 2) when u_1 has an attack range long enough so that when it attacks u_2, it has time to attack, turn around and escape before it is in range of the attack of u_2. The exact conditions under which those conditions hold might be complex to assess, since they depend on many low-level details of the unit dynamics. However, in general, we can define two simple conditions that, if satisfied ensure that we can perform a successful kiting behavior:

$$u_1.speed > u_2.speed \tag{4.1}$$

$$u_1.attackRange > u_2.attackRange + u_2.speed \times kitingTime(u_1) \tag{4.2}$$

where $kitingTime$ represents the time that u_1 requires to decelerate, turn around, attack, turn

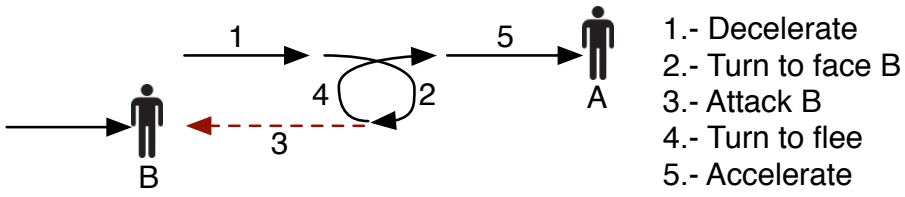


Figure 4.13: The five actions required when a unit A kites another unit B.

around again and leave:

$$kitingTime(u) = u.deceleration + u.attackTime + u.acceleration + 2 \times u.turn$$

Condition 4.1 ensures that u_1 can increase its distance from u_2 if needed. Condition 4.2 ensures that u_1 can attack u_2, turn around and escape, before being inside of the attack range of u_2. In other words, u_1 can attack and retreat before the enemy unit u_2 can attack. Figure 4.13 illustrates each of the different actions that consume time during a kiting movement. Notice that in some complex RTS games, such as StarCraft, the attributes of the different units are not constant, but can be affected dynamically by some of the unit's special abilities during the game. For example, Marines (a basic unit in the game StarCraft) can use an ability called "stimpack" in order to increase their *speed* and decrease their *attackTime* for a brief period of time. Thus, the previous equations constitute just a simplification that might need to be adapted for the specific game at hand.

For example, in the case of StarCraft, the previous equations are satisfied when u_1 is of the type *Vulture* (a fast and versatile ranged Terran unit) and u_2 is of the type *Zealot* (a strong close-range ground Protoss unit). Table 4.7 shows the attributes of both Vultures and Zealot units.

Influence Maps for Kiting

An *Influence Map* is a technique to store relevant information about different entities of interest in a game, represented as a numerical *influence*. In the context of RTS games, an influence map can be seen as a two dimensional matrix containing one cell per each position in the game map. An example can be seen on Figure 4.14, where the influence map contains numerical influence from a unit and from some walls in the map. Influence maps are closely related to the idea of potential fields [170], sharing the same underlying principles.

We can use the abstraction of an influence map to keep track of the dangerous areas (where friendly units will be in range of fire of enemy units) or the safe ones. The main idea in our approach is to use influence maps to detect which areas are safe, then, when performing kiting, the friendly unit will first attack the target unit, and then flee to a safe position using the influence map. Once a safe position has been reached, the kiting behavior will be repeated.

An important decision to be made while designing influence maps is the spatial partition. An influence map with a high resolution can be computationally non-viable on real-time environments, but a low resolution one may provide too coarse grained information, making it useless for taking decisions. In our experiments, we keep three different game maps at different resolutions: a pixel level map for keeping unit location, a walk tile map (with a resolution of 8×8 pixels) for collision detection, and a build tile map (with a resolution of 32×32 pixels) for determining where new buildings can be placed. In our case we decided to use build tile resolution for influence maps since all units are bigger than 8×8 pixel size, and thus this resolution is enough for kiting.

For the purposes of kiting, each unit u_1 performing kiting will compute an influence map. We are interested only on storing two pieces of information in our influence maps: whether a cell is inside of the range of fire of an enemy, and whether a cell is too close to a wall and a unit might run the risk of getting trapped. Those two pieces of information are stored in the influence map in the following way.

- **Enemy units:** We assigned an influence field to each enemy u_2 with a constant influence

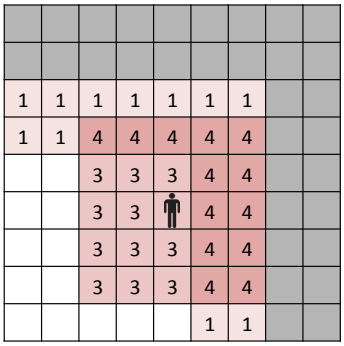


Figure 4.14: Example of Influence Map. The numbers are the threat in that position, a higher value means more danger.

value based on the distance in Eq. 4.2. Then we use Eq. 4.3 to define the maximum distance $d_{max}(u_1, u_2)$ of the field on Eq. 4.4:

$$d_{max}(u_1, u_2) = u_2.attackRange + k + u_2.speed \times kitingTime(u_1) \tag{4.3}$$

Where k is a confidence constant value to ensure a safe distance. In our case we established this constant with a value of 1. The value added to the influence map to a position at distance d from an enemy u_2 is defined as:

$$I_{enemy}(u_1, u_2, d) = \begin{cases} u_2.DPS & \text{if } d \leq d_{max}(u_1, u_2) \\ 0 & \text{if } d > d_{max}(u_1, u_2) \end{cases} \tag{4.4}$$

- **Walls:** One of the problems on a kiting movement is getting stuck on a corner or in a closed area. In order to avoid this, we define an influence field on each wall.

$$I_{wall}(d) = \begin{cases} 1 & \text{if } d \leq 3 \\ 0 & \text{if } d > 3 \end{cases} \tag{4.5}$$

Where 3 is the radius of the field for walls, which was determined empirically.

We can see an example of generated Influence Map on Figure 4.14. The influence map defined in this section can be used by a unit to flee while performing kiting. The next section describes how to decide which of the enemy units to attack when performing kiting.

Target Selection

Selecting the right target is essential to maximize the effectiveness of kiting. In our approach, units select a target by assigning a score to each enemy unit, and then selecting the unit with the maximum score. The score is based on three factors: distance to target, tactical threat and aggro.[3]

- **Distance:** the pixel distance to the target.
- **Tactical threat:** a fixed, hand assigned, value to each different unit type depending on its features and abilities.
- **Aggro:** the result of the DPS (Damage Per Second) that would be inflicted to the target by the attacking unit divided by the time to kill the target.

[3] "aggro" is a slang term meaning "aggravation" or "aggression". In RTS and role-playing games, aggro denotes the aggressive interests of a unit.

Algorithm 4 High level algorithm to perform kiting

```
1: function TICK( )
2:     target ← TARGETSELECTION( )
3:     if CANKITE(target) then
4:         KITINGATTACK(target)
5:     else
6:         ATTACK(target)
7:
8: function KITINGATTACK(target)
9:     position ← GETSECUREPOSITION(actualPos)
10:    if target = actualPos then
11:        ATTACK(target)
12:    else
13:        MOVE(position)                                  ▷ flee movement
```

We can calculate the *aggro* of unit u_1 for unit u_2 in the following way:

$$aggro(u_1, u_2) = \frac{u_2.DPS \text{ to } u_1}{\text{time for } u_1 \text{ to kill } u_2} \tag{4.6}$$

$$\text{time for } u_1 \text{ to kill } u_2 = \frac{u_2.HP}{u_1.DPS \text{ to } u_2} \tag{4.7}$$

The score of an enemy unit is computed as the weighted sum of the three factors:

$$Score(u) = aggro(u) \times w_1 + tactical(u) \times w_2 + distance(u) \times w_3 \tag{4.8}$$

By default NOVA gives a lot of importance to aggro since this variable indicates the most dangerous units that you can kill more quickly. But this can lead in a wrong target selection on kiting behavior; skipping the closest units to attack the one with the highest score, exposing our unit to enemy fire as a consequence. To solve this issue, in order to successfully perform kiting, the vale of w_3 needs to be high enough to make the distance the most significant factor.

Kiting Algorithm

Algorithm 4 shows the kiting movement algorithm, where TARGETSELECTION returns the best available target for the current unit using Eq. 4.8; CANKITE uses Eqs. 4.1 and 4.2 to check if the current unit can kite the enemy unit selected; and GETSECUREPOSITION gets the unit's actual position and returns the closest secure position on the Influence Map. Then if we are in a secure position we can attack the target, otherwise the unit must keep fleeing.

In practice, it is not necessary to create a different influence map for each friendly unit performing kiting, since all units of the same type can share the same influence map.

4.3.2 Experimental Evaluation

To evaluate our approach, we perform three different experiments in the domain of STARCRAFT. We modify our bot NOVA to incorporate our proposed kiting algorithm to control the Vulture units (since they are a good candidate to perform the kiting behavior due their fast movement). The goal of the first two experiments is to evaluate kiting in isolation by pitting Vultures against groups of opponent units. The third experiment evaluates the impact of kiting in the overall performance of NOVA by doing full-game experiments.

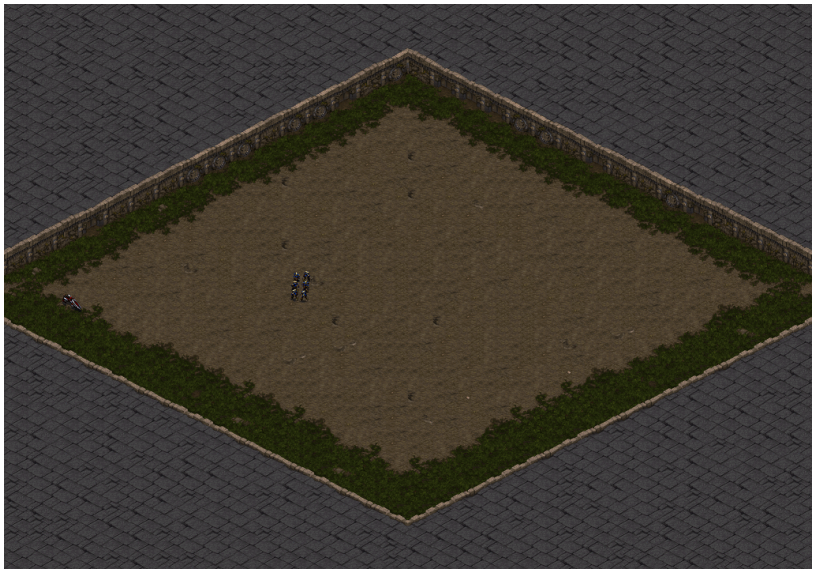


Figure 4.15: Map to test our proposed kiting behavior

Table 4.8: Win ratio on each experiment and setting

Experiment	Setting			
	1	**2**	**3**	**4**
1	0.0%	24.9%	85.5%	95.2%
2	0.0%	98.8%	100.0%	100.0%
3	17.6%	-	-	96.0%

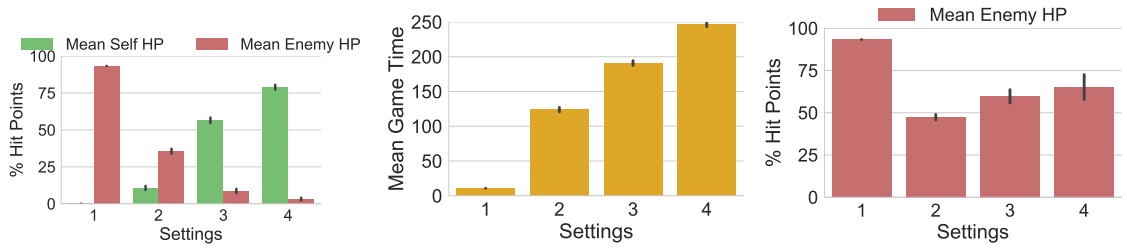
Experiment 1: One-Unit Kiting

In this experiment we show the performance increment that each influence field and the improved target selection brings. We run the experiments in an open square map with one Vulture (our unit) against six Zealots (controlled by the built-in StarCraft AI), shown in Figure 4.15. We configured Nova in four different ways in order to evaluate all the different aspects of our kiting approach:

- **Setting 1:** Default behavior. Here we use the default attack movement of Nova, *i.e.*, when we order a unit to attack a target, that unit will keep attacking without any flee movement.
- **Setting 2:** Influence Map (Enemy field). In this case we execute a kiting movement only assigning charges on enemy units.
- **Setting 3:** Influence Map (Enemy and Wall fields). Here we execute a kiting movement assigning charges on enemy units and map walls.
- **Setting 4:** IM and target selection. Same as previous scenario but now we use our proposed target selection method. This is the full kiting behavior approach presented in this paper.

Our hypothesis is that each successive configuration is better than the previous one. Table 4.8 shows the win ratio of Nova controlling only one Vulture against four Zealots after executing the experiment 1000 times on each setting, showing that kiting drastically increases the performance of Nova, from losing all the games, to winning 95.2% of the time.

Figure 4.16a shows how in each successive setting our final HP is higher than in the previous one, meaning that each setting increases the performance of the algorithm. However, if we look at

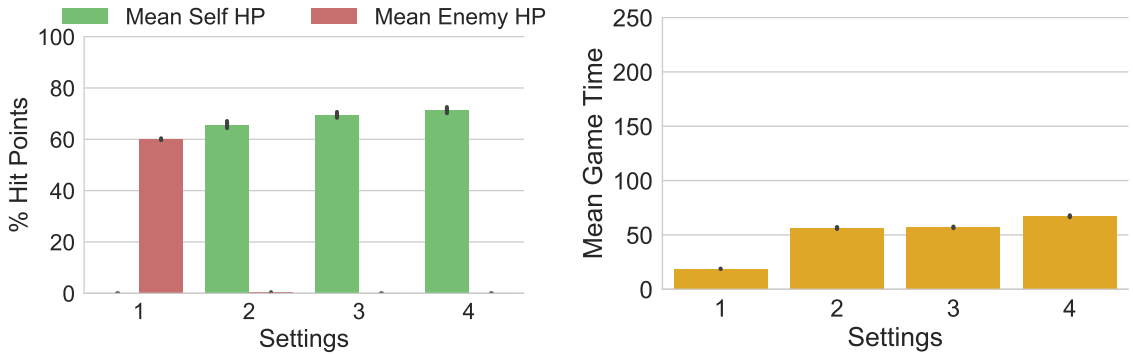


(a) Mean HP and 95% CI of remaining units at the end of the game.

(b) Mean game time and 95% CI.

(c) Mean enemy's HP and 95% CI on games lost.

Figure 4.16: Experiment 1: Results for each of the different settings.



(a) Mean % HP and 95% CI at the end of the game.

(b) Mean game time and 95% CI.

Figure 4.17: Results of Experiment 2.

the enemy's HP from the games we lost (Figure 4.16c) its HP also goes up. The reason is because on each configuration Nova is more cautious than in the previous one, spending more time fleeing than attacking. This is also reflected on game time, showing in Figure 4.16b that games using kiting take more time to resolve because the fleeing movement of Nova's units.

Experiment 2: Group Scalability

Experiment 2 is designed to test how does our kiting behavior scale to groups of units and test whether the two fields we defined are enough. The settings are the same as the previous experiment, with the exception that this time Nova controls four Vultures and the built-in StarCraft AI six Zealots.

During the experiments we observed that units did not disturb each other and the influence map with both fields (enemy and wall) was enough to win 100.0% of the games (as shown in Table 4.8). Figure 4.17a shows that the HP at the end of the game are high even for simpler settings of Nova; and as expected, the time required to win a game (Figure 4.17b) is dramatically reduced compared to Experiment 1.

Experiment 3: Full Game

The previous experiments test our proposed kitting algorithm in a small melee scenario. Experiment 3 aims at evaluating the utility of kiting in a full game. Therefore, here we evaluate the performance of Nova against the built-in Protoss AI of StarCraft. The games are performed in a StarCraft map call Benzene (used in StarCraft AI competitions), and only two of the presented settings (settings 1 and 4) are tested.

Table 4.8 shows the average win ratio of 1000 games with the selected settings. The experiment shows that Nova wins 96 % of the games using our proposed kiting behavior. This is a big improvement if we consider that Nova only wins 17.6 % of the times when not using kiting.

In conclusion, the experiments show that our kiting approach is computationally tractable to use in real-time domains, and it helps the units to survive longer and win more battles. This technique has also a noticeable improvement in the overall performance of StarCraft bots.

Chapter 5: Testing Environment

While we used μRTS for some of our experiments, the main test bed is StarCraft. Since StarCraft is a commercial game (Activison Blizzard), we do not have access to the source code in order to incorporate our algorithms. Therefore, we use the *Brood War Application Programming Interface* (BWAPI) that modifies the game memory used by StarCraft to inject commands. In other words, BWAPI is a framework to interact with StarCraft to build non-cheating AIs or bots.

In this chapter we introduce our StarCraft bot called Nova, a set of StarCraft map scenarios to test algorithms, and we introduce some available tournaments to test our overall bot performance.

5.1 Nova

Nova[1] is our AI agent capable of playing StarCraft using the Terran race. A complete detail of its architecture is described in my master's thesis [131], this section summarizes the most important ideas.

Nova uses a multi-agent system architecture (shown in Figure 5.1) with two types of agents: regular *agents* and *managers*. The difference between regular agents and managers is that Nova can create an arbitrary number of regular agents of each type, but only one manager of each different type. The different agents and managers shown in Figure 5.1 have the following function:

- The **Information Manager** is the environment perception of the bot, it retrieves the game state from the game and stores all the relevant information in a shared blackboard. Uses BWTA2 to analyze the map and store enemy aware information in several influence maps.
- The **Strategy Manager**, based on the data from the Information Manager, determines the high level strategy and sends orders to other managers (Build Manager, Planner Manager, and Squad Manager) in order to execute it. It uses a simple Finite State Machine to transition between strategy goals.
- The **Build Manager** is in charge of looking for the best spot for new buildings to create in its building queue. It automatically replaces destroyed buildings.
- The **Worker Manager** is in charge of controlling all the workers and balance their tasks such as harvesting the different types of resources, or performing scouting tasks.
- The **Planner Manager** is in charge of producing the necessary military units, deciding which are the next units to be produced.
- The **Production Manager** balances the production of new units and research amongst the different buildings capable of performing such tasks.
- The **Squad Manager** is in charge of commanding all the military units to attack the opponent. In order to do so, the Squad Manager spawns one Squad Agent per each group of military units. The orders are decided using the presented Informed MCTSCD.
- The **Squad Agent** controls the group behavior of one group of military units, assigning them a common target and ensuring they move together as a group with steering behaviors like cohesion.
- The **Combat Agent** controls the low level actions of one military unit, including using the special abilities.

[1] `https://bitbucket.org/auriarte/nova`

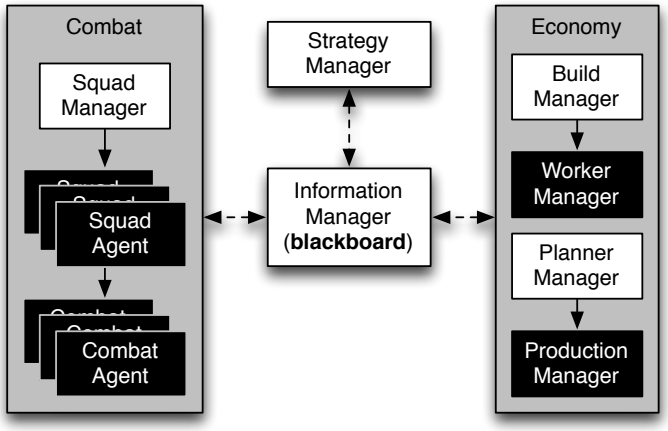


Figure 5.1: Nova multi-agent architecture

5.2 Benchmark Scenarios

In order to assess progress and compare the strengths and weaknesses of different techniques, several competitions (beginning with the ORTS competition and culminating with the current StarCraft AI competitions (Section 5.3), were created in order to provide a common testbed. However, due to the multifaceted nature of RTS games, the results coming out of such competitions are insufficient for understanding the strengths and weaknesses of specific techniques or agents. For example, a StarCraft bot could potentially win a competition by employing a hard-coded strategy that no other bot had a counter-for, while it is clear that finding a good hard-coded strategy for a single specific game (StarCraft) does not contribute toward solving the general problem of AI for RTS games.

We propose a new method of evaluation and comparison of the performance of StarCraft playing agents. We present a benchmark composed of a series of scenarios, each of them capturing a different aspect of RTS games. Each scenario is defined by a starting situation in StarCraft where the agent needs to either defeat the opponent or survive for as long as possible. Additionally, we provide an evaluation metric for each scenario, in order to assess the performance of a given bot in the given scenario. All the scenarios were created for the RTS game StarCraft: Brood War, and these scenarios as well as the necessary scripts to automatically run them and obtain the evaluation metrics are freely available online.

5.2.1 Metrics

In order to assess how well a given agent has addressed a given scenario, we need proper evaluation metrics. Moreover, different scenarios require different metrics. For example, while in some scenarios we are interested in measuring whether the agent was able to completely destroy the opponent, in other scenarios, we might be interested in measuring how long the agent is able to survive. This section presents a collection of metrics that we later use in the definition of the proposed scenarios. We will always assume that the metric is evaluated in a game state S, played by two players, A and B, where A is the player controlled by the agent and B is the opponent. Also, we will use t to denote the amount of time it took for agent A to complete the scenario (i.e., to win, lose, or reach a predefined timeout). All the metrics are designed to be normalized either in the interval [0,1] or [-1,1], with higher values representing better agent performance.

Survivor's life: To evaluate the performance of a *reactive control* technique, the typical scenario is a small map with two military forces fighting each other. In this case we want to win while losing as few units as possible and in the least time possible. For this reason we suggest a modified version of LTD2 (Life-Time Damage 2) proposed by Kovarsky and Buro [132] where

we only take into account the hit points of the surviving units and the time elapsed. Therefore, we compute the survivor's life metric (SL) as the sum of the square root of hit points remaining of each unit divided by amount of time it took to complete the scenario (win/defeat/timeout), measured in frames:

$$SL(S) = \frac{\sum_{a \in U_A} \sqrt{HP(a)} - \sum_{b \in U_B} \sqrt{HP(b)}}{t}$$

where $HP(u)$ is the Hit Points of a unit u, U_X is the set of units of the player X. Moreover, in order to be able to compare results obtained in different scenarios, we normalize the result to the interval $[-1, 1]$. To do this we need to compute the lower and upper bounds. The lower bound is when player A is defeated in the minimum time and without dealing any damage to player B:

$$timeToKill(A, B) = \frac{\sum_{a \in U_A} HP(a)}{\sum_{b \in U_B} DPF(b)}$$

$$lowerBound(S) = \frac{-\sum_{b \in U_B} \sqrt{HP(b)}}{timeToKill(A, B)}$$

where the DPF is the Damage Per Frame a unit can inflict. The upper bound can be computed analogously by considering the situation when player B is defeated in the minimum possible time. Once the bounds are computed, if the SL is negative, we normalize using the lower bound. Otherwise, we use the upper bound to normalize:

$$SL^{norm}(S) = \begin{cases} \frac{SL(S)}{|lowerBound(S)|} & \text{if } SL(S) \leq 0 \\ \frac{SL(S)}{|upperBound(S)|} & \text{otherwise} \end{cases}$$

Time survived: In other scenarios we want to evaluate the amount of time the agent can survive in front of an invincible opponent. We can generate a score between $[0, 1]$ by normalizing the time the agent survived by a predefined timeout (TO): $TS = t/TO$

Time needed: In other scenarios we want to evaluate the time to complete a certain task or to recover from a loss. In this case we start a timer when a certain event happens (e.g., a building is destroyed) and we stop it after a timeout (TO) or after a condition is triggered (e.g., the destroyed building is replaced). The score is computed as: $TN = (TO - t)/TO$.

Units lost: Sometimes, we want to evaluate how a bot can respond to a strategic attack. In these cases, we can count the difference in units lost by players A and B. We can normalize between $[0, 1]$ by dividing the number of units lost by the maximum units of the player:

$$UL = \frac{B_{lost}}{B_{max}} - \frac{A_{lost}}{A_{max}}$$

Where, A_{lost} and B_{lost} are the units that player A and player B lost during the scenario respectively, and A_{max} and B_{max} are predefined upper bounds on the expected number of units each player could lose in a given scenario (this is analogous to a time out in scenarios controlled by time, so if player A loses more than A_{max} units, the scenario ends).

Finally, some parts of StarCraft are stochastic and some AI techniques include stochastic elements. As such, each scenario should be repeated sufficiently large number of times in order to calculate an average value of these metrics that is representative of the bot's performance. All the experiments below show the average and standard deviation after running each scenario 10 times.

5.2.2 Benchmark Scenarios

In this section we present a wide range of scenarios for benchmark purposes. The list is not exhaustive, but is to be seen as a starting point. We expect to keep the number of tests growing. We choose

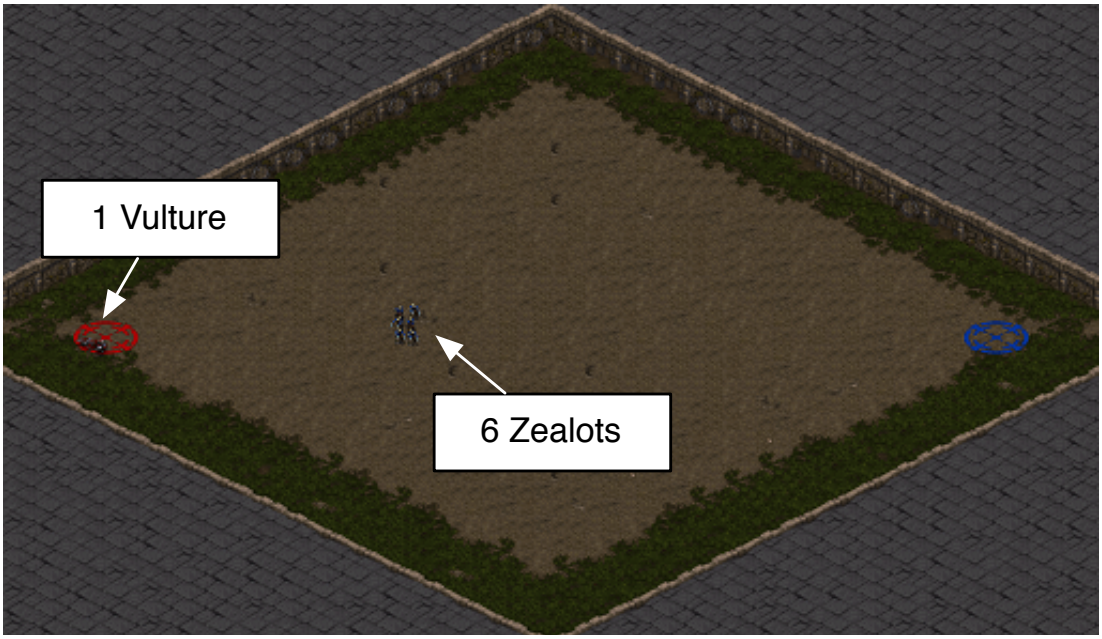


Figure 5.2: StarCraft scenario RC1-A-V6Z.

StarCraft to implement the scenarios because of its recent popularity over other RTS games as a test bed for AI game research [17]. All the scenarios described in this section for StarCraft are freely available online[2] to the research community.

In order to evaluate a StarCraft playing agent using these scenarios, it must support two types of behavior:

- In a micromanagement scenario (those where the player only controls military units) the goal for the intelligent agent should be to reach the starting position of the enemy. Those scenarios are designed in such a way that an enemy confrontation is unavoidable if the intelligent agent tries to reach the opponent's starting position.
- In full-game scenarios, the goal is just like a regular game: do whatever is necessary to win the game.

The following subsections present a detailed list of all the proposed scenarios. For each scenario, we provide the motivation, the map layout, the opponent behavior, termination condition and evaluation metric.

RC1: Reactive Control - Perfect Kiting

The purpose of this scenario is to test whether the intelligent agent is able to reason about the possibility of exploiting its mobility and range attack against a stronger but slower unit in order to win. In this scenario, a direct frontal attack will result in losing the combat, but via careful maneuvering, it is possible to win without taking any damage. Previous works have used this type of scenario to evaluate reactive control of their bots [52, 63, 171, 172]. The different configurations are set to test the scalability and the awareness of the surroundings of the agent.

- **Map description:** We propose two different maps and four initial army configurations. This gives rise to 8 different scenario configurations (**RC1-A-VZ**, **RC1-A-V6Z**, ..., **RC1-B-V6Zg**).

[2]Each scenario consists of a StarCraft map containing the initial situation plus a tournament module to run the benchmark. We provide versions for BWAPI 3.7 and 4.1 downloadable from: `https://bitbucket.org/auriarte/starcraftbenchmarkai`

Layout A: Small map of 64 × 64 tiles with no obstacles. Player A starts at 9 o'clock, player B at 3 o'clock.

Layout B: A big region connected to one small region on each side. Player A starts at the west small region, player B at the east small region. In this layout, intuitively, the player should avoid conflict inside a small region where its troops could get stuck.

VZ: Player A has 1 ranged fast weak unit (Vulture) and player B has 1 melee slow strong unit (Zealot).

V6Z: Player A has 1 ranged fast weak unit (Vulture) and player B has 6 melee slow strong units (Zealot).

3V6Z: Player A has 3 ranged fast weak units (Vulture) and player B has 6 melee slow strong units (Zealot). The purpose of this configuration is to test whether player A's units disturb each other.

V9Zg: Player A has 1 range fast weak unit (Vulture) and player B has 9 melee fast weak units (Zergling).

Figure 5.2 shows an example of scenario RC1-A-V6Z using the **Layout A** with 1 Vulture against 6 Zealots, and Figure 5.3 shows **Layout B** on the scenario RC1-B-V6Z.

- **Enemy behavior:** Player B will try to reach the player A's starting point. If it finds any player A's troops, it will chase it trying to kill it.
- **Termination condition:** One player's army is completely destroyed.
- **Evaluation:** *Survivor's life.*

RC2: Reactive Control - Kiting

In this scenario the intelligent agent is at a disadvantage, but using a hit-and-run behavior might suffice to win. The main difference with the previous case is that here, some damage is unavoidable. Some previous works done in this area in addition to the ones already presented in the previous scenario are [173, 174].

- **Map description:** Here we have two possible configurations:

  A-3D3Z: Using the same map **Layout A** as before, but where player A has 2 Dragoons in his starting locations and 1 Dragoon near player B starting locations, while player B has 3 Zealots. Intuitively, in this configuration player A should retreat his isolated Dragoon in order to wait for reinforcements.

  A-2D3H: Using map **Layout A** where player A has 2 Dragoons and player B has 3 Hydralisks.

- **Enemy behavior:** Player B will try to reach the player's starting point. If it finds any player troop, it will chase it trying to kill it.
- **Termination condition:** One player's army is completely destroyed.
- **Evaluation:** *Survivor's life.*

RC3: Reactive Control - Sustained Kiting

In this case there is no chance to win so we should try to stay alive as much time as possible. A typical example of this behavior is while we are scouting the enemy base. Previous work: [117]

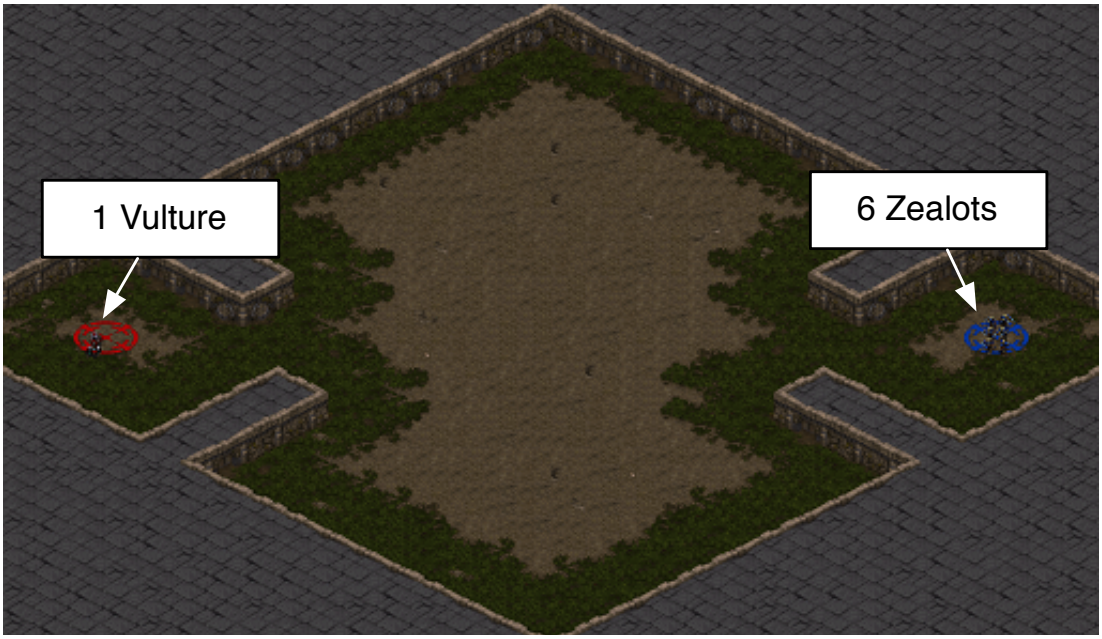


Figure 5.3: StarCraft scenario RC1-B-V6Z.

- **Map description:** This scenario uses a map **Layout C** consisting in two regions (R_1, R_2) connected by a chokepoint. Player A starts in region R_1, which is empty. Player B starts in region R_2 which it has 9 mineral patches and a Vespene gas geyser (like a regular StarCraft starting position); Player B has already a Nexus, 2 Pylons, 2 Zealots and 5 Probes; player A has 1 SCV.
- **Enemy behavior:** Player B will hold its position. If an enemy is within its range of vision, it will chase it with its two Zealots.
- **Termination condition:** One player's army is completely destroyed or timeout after 300 seconds.
- **Evaluation:** *Time survived* in frames since a Zealot starts chasing the SCV normalized by the timeout. Notice that this is a micromanagement map, and thus, the goal of the agent is to reach B's starting position.

RC4: Reactive Control - Symmetric Armies

In equal conditions (symmetric armies), positioning and target selection are key aspects that can determine a player's success in a battle. This scenario presents a test with several configurations as a baseline to experiment against basic AI opponents. Some similar tests have been performed in the past [71, 74, 133, 142, 172, 175–179].

- **Map description:** It combines **Layout A** with the following possible army configurations:

  5V: Each player has 5 Vultures. The goal is to test range units.

  9Z: Each player has 9 Zealots. The goal is to test melee units.

  12D: Each player has 12 Dragoons.

  12Mu: Each player has 12 Mutalisks. To test flying units with "splash" damage.

  20Ma8Me: Each player has 20 Marines and 8 Medics. To test mixed groups.

  5Z8D: Each player has 5 Zealots and 8 Dragoons.

Figure 5.4: StarCraft scenario S1.

- **Enemy behavior:** Player B will hold its position in order to maintain its formation.
- **Termination condition:** One player's army is completely destroyed.
- **Evaluation:** *Survivor's life.*

T1: Tactics - Dynamic obstacles

This scenario measures how well an agent can navigate when chokepoints are blocked by dynamic obstacles (e.g., neutral buildings). Notice that we are not aiming to benchmark pathfinding, but high-level navigation.

- **Map description:** Here we use a professional StarCraft map called Heartbreak Ridge.[3] This map has the particularity that the third closest base location from the starting point has two ways to access it. The first is through an open chokepoint. Alternatively there is a closer chokepoint blocked by an indestructible building and a stack of minerals. The default StarCraft pathfinding algorithm does not take into account dynamic objects, so it will lead the unit through the shortest (blocked) path until the blocking objects are revealed form the fog of war, at which point it will recalculate the path with the new information. Player A starts with a worker in the regular starting position of the map, while the starting position for Player B (who has no units) is this third base expansion. Notice that since this is a micromanagement map, even if B does not have any units in this scenario, the goal of the agent is to reach B's position.
- **Enemy behavior:** None.
- **Termination condition:** Player A reaches the starting position of Player B or a timeout after 60 games seconds.
- **Evaluation:** *Time needed.*

[3] `http://wiki.teamliquid.net/starcraft/Heartbreak_Ridge`

S1: Strategy - Building placement

One of the possible strategies in StarCraft is to *rush* your enemy. *Rushing* consists of training a certain number of military units as fast as possible, then attacking your enemy while she is still developing her economy. A typical counter-strategy to a rush is to create a wall in the chokepoint of your initial region to prevent your opponent from reaching you (a.k.a. walling). This scenario simulates a Zealot rush and is designed to test whether the agent will be able to stop it (intuitively, it seems the only option is to build a wall). Some previous research has been done [92, 93].

- **Map description:** It uses **Layout C** where player A starts like a regular game (a Command Center and 4 SCVs); and player B starts with 4 Zealots. Figure 5.4 shows an example of this scenario using the **Layout C**.
- **Enemy behavior:** Controls player B and it will wait 250 seconds to launch an attack to player A.
- **Termination condition:** Player A loses 25 units (at this point, you are not able to recover from a rush) or player B loses all his 4 Zealots.
- **Evaluation:** In this case we use *units lost* metric. More specifically: (Units player B lost / 4) - (units player A lost / 25).

S2: Strategy - Plan Recovery

An agent should adapt on plan failures. This scenario tests if the AI is able to recover from the opponent disrupting its build order.

- **Map description:** It uses **Layout C** where player A starts like a regular game (a Command Center and 4 SCVs); and player B does not have units.
- **Enemy behavior:** Once player A's Refinery is finished it will be destroyed after 1 second by a scripted event.
- **Termination condition:** When the destroyed building is replaced or a timeout of 400 seconds.
- **Evaluation:** *Time spent* to replace a building normalized by the timeout.

5.2.3 Experiments

In order to illustrate the advantages of the proposed benchmark, we used it to evaluate the performance of some of the state-of-the-art bots available: **FreScBot**, the winner bot of micromanagement tournament at AIIDE 2010; **UAlbertaBot**, winner of AIIDE 2013; **Skynet**, winner of AIIDE 2011, 2012, and CIG 2011, 2012, 2013; and Nova, with a strong kiting behavior. Table 5.1 shows the average score and standard deviation of each bot on each scenario after 10 executions. Not all bots can play all 3 races, and some bots cannot play micromanagement maps. When a bot cannot play a given scenario, we mark it with the N/A (Not Applicable) label.

For scenarios whose metric is normalized in the interval [-1,1], negative values mean that the bots were not able to complete the scenario. At first glance, we can see that most scores are negative, meaning that bots fail to succeed in completing most of the scenarios. We would like to point out a few interesting trends from the table:

- Nova's strong point is kiting, and it is no surprise that it is the only bot that scores well in the kiting scenarios (RC1, RC2, and RC3). However, since Nova can only control the Terran race, it cannot be tested in RC2.
- Newer bots (**UAlbertaBot**, **Skynet**, Nova) significantly outperform the older bot **FreScBot** in micromanagement, even though **FreScBot** was the winner of the micromanagement tournament in 2010. While **UAlbertaBot** and **Skynet** improved the micromanagement of Protoss' units, Nova outperformed **FreScBot**'s micromanagement with Terran's units.

Table 5.1: Score achieved by four different bots on each scenario. The right-most column shows the range of values achievable in each scenario (higher is always better).

Scenario	FreScBot	UAlbertaBot	Skynet	Nova	Range
RC1-A-VZ	-0.160 ± 0.024	-0.222 ± 0.027	-0.243 ± 0.043	0.334 ± 0.052	[-1,1]
RC1-A-V6Z	-0.031 ± 0.012	-0.069 ± 0.008	-0.083 ± 0.010	0.012 ± 0.027	[-1,1]
RC1-A-3V6Z	-0.092 ± 0.033	-0.112 ± 0.026	-0.112 ± 0.028	0.094 ± 0.060	[-1,1]
RC1-A-1V9Zg	-0.018 ± 0.006	-0.044 ± 0.009	-0.036 ± 0.004	0.009 ± 0.038	[-1,1]
RC1-B-VZ	-0.179 ± 0.016	-0.141 ± 0.016	-0.187 ± 0.014	0.289 ± 0.014	[-1,1]
RC1-B-V6Z	-0.059 ± 0.002	-0.044 ± 0.002	-0.064 ± 0.005	0.016 ± 0.025	[-1,1]
RC1-B-3V6Z	-0.118 ± 0.022	-0.089 ± 0.015	-0.109 ± 0.008	0.107 ± 0.037	[-1,1]
RC1-B-1V9Zg	-0.046 ± 0.004	-0.028 ± 0.005	-0.036 ± 0.006	0.028 ± 0.030	[-1,1]
RC2-A-2D3H	-0.234 ± 0.077	0.013 ± 0.126	0.051 ± 0.164	N/A	[-1,1]
RC2-A-3D3Z	0.004 ± 0.076	0.072 ± 0.145	0.289 ± 0.035	N/A	[-1,1]
RC3	N/A	N/A	N/A	0.034 ± 0.007	[0,1]
RC4-A-5V	-0.037 ± 0.013	-0.008 ± 0.019	-0.059 ± 0.029	-0.030 ± 0.067	[-1,1]
RC4-A-12D	-0.003 ± 0.061	0.012 ± 0.035	0.103 ± 0.024	N/A	[-1,1]
RC4-A-9Z	-0.023 ± 0.022	-0.012 ± 0.034	0.024 ± 0.043	N/A	[-1,1]
RC4-A-12Mu	0.104 ± 0.072	0.234 ± 0.060	0.327 ± 0.086	N/A	[-1,1]
RC4-A-20Ma8Me	-0.015 ± 0.031	0.042 ± 0.035	-0.002 ± 0.001	-0.128 ± 0.017	[-1,1]
RC4-A-5Z8D	-0.040 ± 0.017	-0.046 ± 0.025	0.031 ± 0.028	N/A	[-1,1]
T1	N/A	N/A	N/A	0.000 ± 0.000	[0,1]
S1	N/A	-1.000 ± 0.000	N/A	-0.742 ± 0.392	[-1,1]
S2	N/A	0.000 ± 0.000	N/A	0.000 ± 0.000	[0,1]

- Despite improved performance in micromanagement, none of the bots tested were able to pass any of the tactics or strategic scenarios. This is consistent with previous observations indicating that the biggest weakness of state-of-the-art StarCraft bots is their high-level reasoning skills (tactic and macro), and not their micromanagement skills.

Additionally, we would like to point out that the proposed benchmark is not to be seen as a way to assess the strength of a bot overall (the StarCraft AI competition is a better way to assess that), but as a way to delve deeper into the specific strengths and weaknesses of different bots, and a way to guide future work in the area.

5.3 Competitions

Another place to test the overall performance of the StarCraft bots are the AI vs AI competitions. These competitions are typically co-located with scientific conferences. But they are not restricted to the research community.

Started in 2010, the **AIIDE** (Artificial Intelligence for Interactive Digital Entertainment) StarCraft AI Competition[4] is the most well known and longest running StarCraft AI Competition. The first edition had four different tournament categories: 1) was a flat-terrain unit micro-management battle, 2) was another micro-focused game with non-trivial terrain, 3) was a tech-limited StarCraft game on a single known map with no fog-of-war enforced, and 4) was the complete game of StarCraft with fog-of-war enforced in a random map chosen from a map pool of five well-known professional maps used in human competitions. Since the second edition, due to a lack of entrants in tournament categories 1-3, only the full game category have been used and a automated tournament-managing software allowed to run more games in a round-robin format. Also, the competition became open-source, in an effort not only to prevent possible cheating, but to promote healthy competition in

[4] `http://www.StarCraftAICompetition.com`

future tournaments by giving newcomers and easier entry point by basing their design off of previous bots.

Another tournament organized at the **CIG** (Computational Intelligence in Games) started as a full game with fog-of-war enable. The main difference with AIIDE was that the map pool was unknown to the participants to avoid bots using hard coded techniques to exploit a specific map. This difference was discarded later and the last editions of the tournament use the same tournament manager software and rules as AIIDE, but with a different public map pool.

In **Student StarCraft AI Tournament** (SSCAIT)[5] matches are automatized and running all the time ranked in a ladder. It uses an Elo rating system suitable for calculating the *relative skill level* of a bot in two-player games. In the Elo system each player has a numerical rating that gets incremented or decremented some points after each game. The amount of points depends on the difference in the ratings of the players. A player will gain more points by beating a higher-rated player than by beating a lower-rated player. This kind of rating system is widely used in games like Chess. Since the bots are updated regularly, it is a good resource to test the performance of new bots against the current state of the art in StarCraft bots. Once a year they organize a tournament with two divisions: one for students (round robin) and another for everyone (elimination bracket after the round robin).

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

Chapter 6: Conclusions

This thesis presented the first steps towards a holistic approach to solve adversarial search in domains with real time constrains, large search spaces, and imperfect information such as RTS games. The contributions can be categorized in three areas: adversarial search, spatial reasoning and performance evaluation of AI agents.

The contributions in the **adversarial search** category are:

1. A study of game state abstractions (Section 3.1) to reduce the state space and the action space. Showing in our experiments a branching factor reduction from 10^{300} to 10^{10}. The abstractions are based on a map decomposition, extracting the mean features of any given map. Making it suitable for any RTS game.

2. A methodology to learn forward models (Section 3.2), more specifically combat models, in cases where we do not have access to the game forward model (like in close source commercial games) but we have access to a large game replay dataset. The forward models presented are able to work with the game abstractions previously presented; and since the dataset used is from human expert games, it is able to capture how efficiently humans control units, predicting the outcome of a battle assuming a professional human control.

3. How to perform adversarial search when (1) the game state is abstracted, (2) we have durative actions and (3) we learned a forward model using an adapted Monte Carlo Tree Search algorithm (Section 3.3).

4. Informed Monte Carlo Tree Search: despite the fact that reducing the branching factor to 10^{10} is a great reduction over the original branching factor, it is still unfeasible to explore all the branches even just once (as most multi-armed techniques require). To solve this, we presented how to use machine learning techniques, extracting the features regardless the map, to inform the search process and guide it towards the most promising nodes (Section 3.4).

5. Finally in Section 3.6, we proposed a set of sampling techniques to generate a believe state in real-time imperfect information games with large game tree depths such as RTS games. The approach presented only decreases the performance of a game tree search algorithm (like MCTS) by 8%-15% compared to having access to the whole game state (cheating).

The **spatial reasoning** contributions are: (1) a map abstraction algorithm to decompose a map into a graph of regions and adjacent regions (Section 4.1); (2) how to find the build placement locations to close a chokepoint with buildings (Section 4.2); and (3) how to perform an attack-and-retreat behavior via influence maps (Section 4.3).

Finally, Section 5.2 presented a set of scenarios to **test the performance** of a StarCraft bot in different tasks. While general tournaments are good to tell us which bot is stronger, the benchmark suite presented aims to get more insights about the performance of a bot in specific tasks.

In summary, this dissertation tried to answer the question of whether it is possible to use adversarial search in real-time domains with large search spaces, large combinatorial branching factors and imperfect information, where no forward model is readily available such as RTS games. The different contributions in this dissertation constitute the first building blocks toward a solution to such problem, indicating that it is indeed possible. By integrating the different proposed pieces (abstraction to reduce the branching factor and state space, learning forward models, using machine learning to guide MCTS and determinization to handle partial observability) we have shown a complete solution to playing RTS games using game tree search. The current approach, however, still relies on an underlying bot to handle several aspects of the game (such as resource gathering), but

there is no theoretical limitation preventing the MCTS approach from handling every single aspect of the game. Studying how the proposed integrated approach scales to handle the complete game as a holistic game-tree search approach to RTS games will be part of our future work.

6.1 Publications

List of publications from this thesis grouped by are of contribution:

- Adversarial Search
  - (2014) Uriarte and Ontañón. Game Tree Search over High-Level Game States in RTS Games. *AIIDE*
  - (2014) Uriarte and Ontañón. High-Level Representations for Game Tree Search in RTS Games. *AIIDE workshop*
  - (2015) Uriarte and Ontañón. Automatic Learning of Combat Models for RTS Games. *AIIDE*
  - (2016) Uriarte and Ontañón. Improving Monte Carlo Tree Search Policies in StarCraft via Probabilistic Models Learned from Replay Data. *AIIDE*
  - (2017) Uriarte and Ontañón. Combat Models for RTS Games. *TCIAIG*
  - (2017) Uriarte and Ontañón. Single Believe State Generation for Partially Observable Real-Time Strategy Games. *CIG*
- Spatial Reasoning
  - (2012) Uriarte and Ontañón. Kiting in RTS Games Using Influence Maps. *AIIDE workshop*
  - (2014) Richoux, Uriarte and Ontañón. Walling in Strategy Games via Constraint Optimization. *AIIDE*
  - (2015) Richoux, Uriarte and Baffier. GHOST: A Combinatorial Optimization Framework for Real-Time Problems. *TCIAIG*
  - (2016) Uriarte and Ontañón. Improving Terrain Analysis and Applications to RTS Game AI. *AIIDE workshop*
- Survey and Benchmark
  - (2013) Santiago Ontañón, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill and Mike Preuss. A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft. *TCIAIG*
  - (2015) Uriarte and Ontañón. A Benchmark for StarCraft Intelligent Agents. *AIIDE workshop*

6.2 Future Work

This dissertation opened the way for a holistic approach to do adversarial search in complex domains such as RTS games. Future work includes:

- Studying the benefits of extending the proposed approach for abstracting the game state to capture geometric or tactic properties of the map or using different unit clustering techniques. We would also like to study how much does this approach generalize to other RTS games, or other classes of games in general.
- The proposed forward model is just an initial approach for tactical combats (battles between armies). It is able to handle heterogeneous armies, but does not capture aspects such as "area of damage" or temporary invincible units (due invisibility). We would like to explore the possibility of expanding our forward models to capture non-combat actions, like gathering resources and building facilities, in order to simulate all type of actions and not only the ones

related to the combat. In summary, we would like to study how to generalize the proposed ideas to expand the aspects of the game mechanics modeled by the forward model, and also to expand the class of games that can be modeled.

- In our experiments, we only used adversarial search to handle tactical combats and movements. We would like to expand this to handle additional aspects of the game (resource gathering, unit training, build order and technology research), and study the performance of the proposed forward models and search strategies as more and more parts of the game are considered.
- We would like to explore the possibility of using information sets in the game tree search to obtain better mixed strategies. In our experiments (Section 3.6.1) we also observed the initial "depression" at the start of the game (Figure 3.18), our hypothesis is that this is correlated with the initial scouting that professional players do in order to mitigate the imperfect information at the beginning of the game. Therefore this can be used to implement a gathering information algorithm that tries to mitigate this "depression".
- We would also like to further study spatial reasoning to capture additional geometric or tactical aspects of the map, which can be later exploited by game tree search or as a featurization for machine learning techniques.
- We have shown that using special purpose search techniques (CSP) to solve specific sub-problems of the game (like walling) can bring great benefits (and recent work in the community has shown this to be the case for other problems such as build order planning [28]). We would like to study which other sub-problems (like build order or target selection) can be handled by other specialized search methods, which of those can be generalized to other class of games, and how solving those can help reducing the complexity of game tree search.
- Finally, an area that has recently received increased attention is the use of machine learning techniques such as Deep Neural Networks to inform game tree search (as AlphaGo proposed). The complexity of RTS games makes this a really hard problem and the current research has been only focused on small combat scenarios or small domains such as μRTS. As part of our future work, we would like to study how to scale this up to handle complete complex RTS games.

Bibliography

[1] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012. doi: 10.1109/TCIAIG.2012.2186810.

[2] Santiago Ontañón. The combinatorial multi-armed bandit problem and its application to real-time strategy games. In *Artificial Intelligence and Interactive Digital Entertainment*. AAAI Press, 2013. URL `https://www.aaai.org/ocs/index.php/AIIDE/AIIDE13/paper/view/7377`.

[3] Michael Johanson. Measuring the size of large no-limit poker games. *arXiv preprint arXiv:1302.7008*, 2013.

[4] Shirish Chinchalkar. An upper bound for the number of reachable positions. *International Computer Chess Association*, 19(3):181–183, 1996.

[5] John Tromp and Gunnar Farnebäck. Combinatorics of go. In *International Conference on Computers and Games*, pages 84–99, 2006.

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

[7] Brett Alexander Harrison. Move prediction in the game of go. *Undergraduate thesis, Harvard University*, 2010.

[8] Nils J Nilsson. *Principles of artificial intelligence*, volume 1211. Tioga publishing company Palo Alto, CA, 1980.

[9] Shigeki Iwata and Takumi Kasai. The Othello game on an $n \times n$ board is PSPACE-complete. *Theoretical Computer Science*, 123(2):329–340, 1994.

[10] Aviezri S Fraenkel and David Lichtenstein. Computing a perfect strategy for $n \times n$ Chess requires time exponential in n. *Combinatorial Theory, Series A*, 32(2):199–214, 1981.

[11] John Michael Robson. The complexity of Go. In *IFIP Congress*, pages 413–417, 1983.

[12] Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. Alternation. *ACM*, 28(1): 114–133, 1981. doi: 10.1145/322234.322243.

[13] John H Reif. The complexity of two-player games of incomplete information. *Journal of computer and system sciences*, 29(2):274–301, 1984.

[14] Giovanni Viglietta. Gaming is a hard job, but someone has to do it! *Theory of Computing Systems*, 54(4):595, 2014. doi: 10.1007/s00224-013-9497-5. URL `http://dx.doi.org/10.1007/s00224-013-9497-5`.

[15] Timothy Furtak and Michael Buro. On the complexity of two-player attrition games played on graphs. In G. Michael Youngblood and Vadim Bulitko, editors, *Artificial Intelligence and Interactive Digital Entertainment*. AAAI Press, 2010. URL `https://www.aaai.org/ocs/index.php/AIIDE/AIIDE10/paper/view/2132`.

[16] Michael Buro. Real-time strategy games: a new AI research challenge. In *International Joint Conference on Artificial Intelligence*, pages 1534–1535. Morgan Kaufmann Publishers Inc., 2003. URL `http://ijcai.org/Proceedings/03/Papers/265.pdf`.

[17] Santiago Ontañón, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill, and Mike Preuss. A survey of real-time strategy game AI research and competition in StarCraft. *IEEE Transactions on Computational Intelligence and AI in Games*, 5(4):293–311, 2013. doi: 10.1109/TCIAIG.2013.2286295.

[18] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529:484–503, 2016. doi: 10.1038/nature16961.

[19] Simon Dor. The heuristic circle of real-time strategy process: A StarCraft: Brood War case study. *Game Studies*, 14(1), August 2014. URL `http://gamestudies.org/1401/articles/dor`.

[20] Glen Robertson and Ian D. Watson. A review of real-time strategy game AI. *AI Magazine*, 35 (4):75–104, 2014. URL `http://www.aaai.org/ojs/index.php/aimagazine/article/view/2478`.

[21] Richard K Guy. *Combinatorial games*, volume 43. AMS Bookstore, 2000.

[22] Daniel Fu and Ryan Houlette. *The Ultimate Guide to FSMs in Games*, volume 2, pages 283–302. Charles River Media, 2004.

[23] 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, 2008. ISBN 978-3-540-77464-8. doi: 10.1007/978-3-540-77465-5_15.

[24] Kinshuk Mishra, Santiago Ontañón, and Ashwin Ram. Situation assessment for plan retrieval in real-time strategy games. In *European Conference on Advances in Case-Based Reasoning ECCBR*, volume 5239 of *Lecture Notes in Computer Science*, pages 355–369. Springer, 2008. doi: 10.1007/978-3-540-85502-6_24.

[25] Hector Muñoz-Avila and David Aha. On the role of explanation for hierarchical case-based planning in real-time strategy games. In *European Conference on Advances in Case-Based Reasoning ECCBR Workshops*, pages 1–10, 2004.

[26] Jasper Laagland. A HTN planner for a real-time strategy game, 2006. URL `http://hmi.ewi.utwente.nl/verslagen/capita-selecta/CS-Laagland-Jasper.pdf`.

[27] Franisek Sailer, Michael Buro, and Marc Lanctot. Adversarial planning through strategy simulation. In *Computational Intelligence and Games*, pages 80–87. IEEE, 2007. doi: 10.1109/CIG.2007.368082.

[28] David Churchill and Michael Buro. Build order optimization in StarCraft. In *Artificial Intelligence and Interactive Digital Entertainment*, pages 14–19. AAAI Press, 2011. URL `https://www.aaai.org/ocs/index.php/AIIDE/AIIDE11/paper/view/4078`.

[29] Ben G. Weber and Michael Mateas. A data mining approach to strategy prediction. In *Computational Intelligence and Games*, pages 140–147. IEEE, 2009. doi: 10.1109/CIG.2009.5286483.

[30] Ethan W Dereszynski, Jesse Hostetler, Alan Fern, Thomas G Dietterich, Thao-Trang Hoang, and Mark Udarbe. Learning probabilistic behavior models in real-time strategy games. In *Artificial Intelligence and Interactive Digital Entertainment*. AAAI Press, 2011. URL `https://www.aaai.org/ocs/index.php/AIIDE/AIIDE11/paper/view/4073`.

[31] Gabriel Synnaeve and Pierre Bessiere. A bayesian model for opening prediction in RTS games with application to StarCraft. In *Computational Intelligence and Games*, pages 281–288. IEEE, 2011. doi: 10.1109/CIG.2011.6032018.

[32] Gabriel Synnaeve and Pierre Bessière. A bayesian model for plan recognition in RTS games applied to StarCraft. In *Artificial Intelligence and Interactive Digital Entertainment*, pages 79–84. AAAI Press, 2011. URL `https://www.aaai.org/ocs/index.php/AIIDE/AIIDE11/paper/view/4062`.

[33] Agnar Aamodt and Enric Plaza. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI communications*, 7(1):39–59, 1994.

[34] David W Aha, Matthew Molineaux, and Marc Ponsen. Learning to win: Case-based plan selection in a real-time strategy game. In *International Conference on Case-Based Reasoning*, volume 3620 of *Lecture Notes in Computer Science*, pages 5–20. Springer, 2005. doi: 10.1007/11536406.

[35] Ji-Lung Hsieh and C-T Sun. Building a player strategy model by analyzing replays of real-time strategy games. In *IJCNN*, pages 3106–3111. IEEE, 2008.

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

[37] Ulit Jaidee, Héctor Muñoz-Avila, and David W Aha. Case-based learning in goal-driven autonomy agents for real-time strategy combat tasks. In *International Conference on Case-Based Reasoning*, pages 43–52, 2011.

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

[39] Matteus Magnusson and Suresh Balsasubramaniyan. A communicating and controllable teammate bot for RTS games. Master's thesis, Blekinge Institute of Technology, 2012. URL `http://www.diva-portal.org/smash/get/diva2:831695/FULLTEXT01.pdf`.

[40] Robin Karlsson. Cooperative behaviors betweentwo teaming RTS bots in StarCraft. Master's thesis, Blekinge Institute of Technology, 2015.

[41] Stephen Hladky and Vadim Bulitko. An evaluation of models for predicting opponent positions in first-person shooter video games. In *Computational Intelligence and Games*, pages 39–46. IEEE, 2008. doi: 10.1109/CIG.2008.5035619.

[42] Froduald Kabanza, Philipe Bellefeuille, Francis Bisson, Abder Rezak Benaskeur, and Hengameh Irandoust. Opponent behaviour recognition for real-time strategy games. *Plan, Activity, and Intent Recognition*, 10, 2010.

[43] Christopher W Geib and Robert P Goldman. A probabilistic plan recognition algorithm based on plan tree grammars. *Artificial Intelligence*, 173(11):1101–1132, 2009.

[44] Manu Sharma, Michael P Holmes, Juan Carlos Santamaría, Arya Irani, Charles Lee Isbell Jr, and Ashwin Ram. Transfer learning in real-time strategy games using hybrid CBR/RL. In *International Joint Conference on Artificial Intelligence*, volume 7, pages 1041–1046, 2007. URL `http://ijcai.org/Proceedings/07/Papers/168.pdf`.

[45] Pedro Cadena and Leonardo Garrido. Fuzzy case-based reasoning for managing strategic and tactical reasoning in StarCraft. In *Mexican International Conference on Artificial Intelligence*, volume 7094 of *Lecture Notes in Computer Science*, pages 113–124. Springer, 2011. doi: 10.1007/978-3-642-25324-9.

[46] Gabriel Synnaeve and Pierre Bessière. Special tactics: A bayesian approach to tactical decision-making. In *Computational Intelligence and Games*, pages 409–416. IEEE, 2012. doi: 10.1109/CIG.2012.6374184.

[47] Luke Perkins. Terrain analysis in real-time strategy games: An integrated approach to choke point detection and region decomposition. In *Artificial Intelligence and Interactive Digital Entertainment*. AAAI Press, 2010. URL `https://www.aaai.org/ocs/index.php/AIIDE/AIIDE10/paper/view/2114`.

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

[49] David Churchill, Abdallah Saffidine, and Michael Buro. Fast heuristic search for RTS game combat scenarios. In *Artificial Intelligence and Interactive Digital Entertainment*. AAAI Press, 2012. URL `https://www.aaai.org/ocs/index.php/AIIDE/AIIDE12/paper/view/5469`.

[50] Michael Chung, Michael Buro, and Jonathan Schaeffer. Monte carlo planning in RTS games. In *Computational Intelligence and Games*. IEEE, 2005. URL `http://csapps.essex.ac.uk/cig/2005/papers/p1051.pdf`.

[51] Radha-Krishna Balla and Alan Fern. UCT for tactical assault planning in real-time strategy games. In *International Joint Conference on Artificial Intelligence*, pages 40–45, 2009. URL `http://ijcai.org/Proceedings/09/Papers/018.pdf`.

[52] Alberto Uriarte and Santiago Ontañón. Kiting in RTS games using influence maps. In *AIIDE Workshop on Artificial Intelligence in Adversarial Real-Time Games*. AAAI Press, 2012. URL `https://www.aaai.org/ocs/index.php/AIIDE/AIIDE12/paper/view/5497`.

[53] Johan Hagelbäck and Stefan J Johansson. A multiagent potential field-based bot for real-time strategy games. *International Journal of Computer Games Technology*, 2009.

[54] Johan Hagelbäck. Potential-field based navigation in StarCraft. In *Computational Intelligence and Games*, pages 388–393. IEEE, 2012. doi: 10.1109/CIG.2012.6374181.

[55] Johan Hagelbäck and Stefan J. Johansson. Dealing with fog of war in a real time strategy game environment. In *Computational Intelligence and Games*, pages 55–62. IEEE, 2008. doi: 10.1109/CIG.2008.5035621.

[56] Phillipa Avery, Sushil Louis, and Benjamin Avery. Evolving coordinated spatial tactics for autonomous entities using influence maps. In *Computational Intelligence and Games*, pages 341–348. IEEE, 2009. doi: 10.1109/CIG.2009.5286457.

[57] Greg Smith, Phillipa Avery, Ramona Houmanfar, and Sushil Louis. Using co-evolved RTS opponents to teach spatial tactics. In *Computational Intelligence and Games*, pages 146–153. IEEE, 2010. doi: 10.1109/ITW.2010.5593359.

[58] Holger Danielsiek, Raphael Stür, Andreas Thom, Nicola Beume, Boris Naujoks, and Mike Preuss. Intelligent moving of groups in real-time strategy games. In *Computational Intelligence and Games*, pages 71–78. IEEE, 2008. doi: 10.1109/CIG.2008.5035623.

[59] Liang Liu and Longshu Li. Regional cooperative multi-agent Q-learning based on potential field. In *International Conference on Natural Computation*, volume 6, pages 535–539, 2008.

[60] Mike Preuss, Nicola Beume, Holger Danielsiek, Tobias Hein, Boris Naujoks, Nico Piatkowski, Raphael Ster, Andreas Thom, and Simon Wessing. Towards intelligent team composition and maneuvering in real-time strategy games. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(2):82–98, 2010.

[61] Gabriel Synnaeve and Pierre Bessiere. A bayesian model for RTS units control applied to StarCraft. In *Computational Intelligence and Games*, pages 190–196. IEEE, 2011. doi: 10.1109/CIG.2011.6032006.

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

[63] Stefan Wender and Ian D. Watson. Applying reinforcement learning to small scale combat in the real-time strategy game StarCraft:Broodwar. In *Computational Intelligence and Games*, pages 402–408. IEEE, 2012. doi: 10.1109/CIG.2012.6374183.

[64] Bhaskara Marthi, Stuart Russell, David Latham, and Carlos Guestrin. Concurrent hierarchical reinforcement learning. In *International Joint Conference on Artificial Intelligence*, pages 779–785. Professional Book Center, 2005. URL `http://ijcai.org/Proceedings/05/Papers/1552.pdf`.

[65] Charles Madeira, Vincent Corruble, and Geber Ramalho. Designing a reinforcement learning-based adaptive AI for large-scale strategy games. In *Artificial Intelligence and Interactive Digital Entertainment*, pages 121–123. AAAI Press, 2006. URL `http://www.aaai.org/Library/AIIDE/2006/aiide06-026.php`.

[66] Ulit Jaidee and Héctor Muñoz-Avila. CLASSQ-L: A q-learning algorithm for adversarial real-time strategy games. In *AIIDE Workshop on Artificial Intelligence in Adversarial Real-Time Games*. AAAI Press, 2012. URL `https://www.aaai.org/ocs/index.php/AIIDE/AIIDE12/paper/view/5515`.

[67] Gabriel Synnaeve, Nantas Nardelli, Alex Auvolat, Soumith Chintala, Timothée Lacroix, Zeming Lin, Florian Richoux, and Nicolas Usunier. Torchcraft: a library for machine learning research on real-time strategy games. *arXiv preprint arXiv:1611.00625*, 2016.

[68] Nicolas Usunier, Gabriel Synnaeve, Zeming Lin, and Soumith Chintala. Episodic exploration for deep deterministic policies: An application to starcraft micromanagement tasks. *arXiv preprint arXiv:1609.02993*, 2016.

[69] Peng Peng, Quan Yuan, Ying Wen, Yaodong Yang, Zhenkun Tang, Haitao Long, and Jun Wang. Multiagent bidirectionally-coordinated nets for learning to play starcraft combat games. *arXiv preprint arXiv:1703.10069*, 2017.

[70] Kshitij Judah, Saikat Roy, Alan Fern, and Thomas G Dietterich. Reinforcement learning via practice and critique advice. In *AAAI Conference on Artificial Intelligence*, pages 481–486. AAAI Press, 2010. URL `http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1882`.

[71] Amirhosein Shantia, Eric Begue, and Marco Wiering. Connectionist reinforcement learning for intelligent unit micro management in StarCraft. In *IJCNN*, pages 1794–1801. IEEE, 2011. doi: 10.1109/IJCNN.2011.6033442.

[72] Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*. University of Cambridge, Department of Engineering, 1994.

[73] Marc Ponsen. *Improving Adaptive Game AI with Evolutionary Learning*. PhD thesis, University of Wolverhampton, 2004.

[74] Gabriel Iuhasz, Viorel Negru, and Daniela Zaharie. Neuroevolution based multi-agent system for micromanagement in real-time strategy games. In *Balkan Conference in Informatics (BCI 2012)*, pages 32–39. ACM, 2012. doi: 10.1145/2371316.2371324.

[75] Nasri Othman, James Decraene, Wentong Cai, Nan Hu, and Alexandre Gouaillard. Simulation-based optimization of StarCraft tactical AI through evolutionary computation. In *Computational Intelligence and Games*. IEEE, 2012. doi: 10.1109/CIG.2012.6374182.

[76] Tomasz Szczepański and Agnar Aamodt. Case-based reasoning for improved micromanagement in real-time strategy games. In *International Conference on Case-Based Reasoning*, pages 139–148, 2009.

[77] In-Seok Oh, Hochul Cho, and Kyung-Joong Kim. Playing real-time strategy games by imitating human players' micromanagement skills based on spatial analysis. *Expert Systems with Applications*, 71:192–205, 2017.

[78] Sven Koenig and Maxim Likhachev. D* lite. In *National Conference on Artificial Intelligence and Conference on Innovative Applications of Artificial Intelligence*, pages 476–483, 2002.

[79] Adi Botea, Martin Mueller, and Jonathan Schaeffer. Near optimal hierarchical path-finding. *Journal of Game Development*, 1(1):1–22, 2004.

[80] Douglas Demyen and Michael Buro. Efficient triangulation-based pathfinding. In *National Conference on Artificial Intelligence and Conference on Innovative Applications of Artificial Intelligence*, pages 942–947. AAAI Press, 2006. URL `http://www.aaai.org/Library/AAAI/2006/aaai06-148.php`.

[81] Daniel Damir Harabor, Alban Grastien, et al. Online graph pruning for pathfinding on grid maps. In *AAAI Conference on Artificial Intelligence*. AAAI Press, 2011. URL `http://www.aaai.org/ocs/index.php/AAAI/AAAI11/paper/view/3761`.

[82] Craig W. Reynolds. Steering behaviors for autonomous characters. In *Game Developers Conference*, 1999.

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

[84] Nathan R. Sturtevant. Benchmarks for grid-based pathfinding. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2):144–148, 2012. doi: 10.1109/TCIAIG.2012.2197681.

[85] Dave C. Pottinger. Terrain analysis for real-time strategy games. In *Game Developers Conference*, 2000.

[86] Kenneth D. Forbus, James V. Mahoney, and Kevin Dill. How qualitative spatial reasoning can improve strategy game AIs. *IEEE Intelligent Systems*, 17(4):25–30, 2002. doi: http://doi.ieeecomputersociety.org/10.1109/MIS.2002.1024748.

[87] D Hunter Hale, G Michael Youngblood, and Priyesh N Dixit. Automatically-generated convex region decomposition for real-time spatial agent navigation in virtual worlds. In *Artificial Intelligence and Interactive Digital Entertainment*, pages 173–178. AAAI Press, 2008. URL `http://www.aaai.org/Library/AIIDE/2008/aiide08-029.php`.

[88] Daniel Higgins. Terrain analysis in an rts - the hidden giant. *Game Programming Gems 3*, pages 268–284, 2002.

[89] Kári Halldórsson and Yngvi Björnsson. Automated decomposition of game maps. In *Artificial Intelligence and Interactive Digital Entertainment*, volume 15, pages 122–127. AAAI Press, 2015. URL `https://www.aaai.org/ocs/index.php/AIIDE/AIIDE15/paper/view/11522`.

[90] W. Bidakaew, P. Sompagdee, S. Ratanotayanon, and P. Vichitvejpaisal. Rts terrain analysis: An axial-based approach for improving chokepoint detection method. In *Knowledge and Smart Technology*, pages 228–233, 2016. doi: 10.1109/KST.2016.7440512.

[91] Chen Si, Yusuf Pisan, and Chek Tien Tan. Automated terrain analysis in real-time strategy games. In *Foundations of Digital Games*, 2014. URL `http://www.fdg2014.org/papers/fdg2014_dc_04.pdf`.

[92] Michal Čertický. Implementing a wall-in building placement in StarCraft with declarative programming. *arXiv preprint arXiv:1306.4460*, 2013. URL http://arxiv.org/abs/1306.4460.

[93] Florian Richoux, Alberto Uriarte, and Santiago Ontañón. Walling in strategy games via constraint optimization. In *Artificial Intelligence and Interactive Digital Entertainment*. AAAI Press, 2014. URL https://www.aaai.org/ocs/index.php/AIIDE/AIIDE14/paper/view/8956.

[94] Florian Richoux, Alberto Uriarte, and Jean-François Baffier. GHOST: A combinatorial optimization framework for real-time problems. *IEEE Transactions on Computational Intelligence and AI in games*, 2016. doi: 10.1109/TCIAIG.2016.2573199.

[95] Caio Freitas de Oliveira and Charles Andrye Galvao Madeira. Creating efficient walls using potential fields in real-time strategy games. In *Computational Intelligence and Games*, pages 138–145. IEEE, 2015. doi: 10.1109/CIG.2015.7317921.

[96] Ulrich Schwalbe and Paul Walker. Zermelo and the early history of game theory. *Games and economic behavior*, 34(1):123–137, 2001.

[97] Harold W Kuhn. A simplified two-person poker. *Contributions to the Theory of Games*, 1: 97–103, 1950.

[98] Daphne Koller and Avi Pfeffer. Generating and solving imperfect information games. In *International Joint Conference on Artificial Intelligence*, pages 1185–1193. Morgan Kaufmann, 1995. URL http://ijcai.org/Proceedings/95-2/Papers/022.pdf.

[99] Ian Frank and David A. Basin. Search in games with incomplete information: A case study using bridge card play. *Artificial Intelligence*, 100(1-2):87–123, 1998. doi: 10.1016/S0004-3702(97)00082-9.

[100] Ian Frank, David A Basin, and Hitoshi Matsubara. Finding optimal strategies for imperfect information games. In *National Conference on Artificial Intelligence and Conference on Innovative Applications of Artificial Intelligence*, pages 500–507. AAAI Press / The MIT Press, 1998. URL http://www.aaai.org/Library/AAAI/1998/aaai98-071.php.

[101] Stuart Russell and Jason Wolfe. Efficient belief-state AND-OR search, with application to Kriegspiel. In *International Joint Conference on Artificial Intelligence*, volume 19, pages 278–285. Professional Book Center, 2005. URL http://ijcai.org/Proceedings/05/Papers/0929.pdf.

[102] Martin Zinkevich, Michael Johanson, Michael H Bowling, and Carmelo Piccione. Regret minimization in games with incomplete information. In *Neural Information Processing Systems*, pages 1729–1736. Curran Associates, Inc., 2007. URL http://papers.nips.cc/paper/3306-regret-minimization-in-games-with-incomplete-information.

[103] Marc Lanctot, Kevin Waugh, Martin Zinkevich, and Michael Bowling. Monte carlo sampling for regret minimization in extensive games. In *Neural Information Processing Systems*, pages 1078–1086. Curran Associates, Inc., 2009. URL http://papers.nips.cc/paper/3713-monte-carlo-sampling-for-regret-minimization-in-extensive-games.

[104] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, 3rd edition, 2009. ISBN 0136042597.

[105] Jeffrey Richard Long, Nathan R Sturtevant, Michael Buro, and Timothy Furtak. Understanding the success of perfect information monte carlo sampling in game tree search. In *AAAI Conference on Artificial Intelligence*. AAAI Press, 2010. URL http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1876.

[106] R. A. Corlett and S. J. Todd. A monte-carlo approach to uncertain inference. In *Artificial intelligence and its applications*, pages 127–137, 1986.

[107] David NL Levy. The million pound bridge program. *Heuristic Programming in Artificial Intelligence*, pages 95–103, 1989.

[108] Ian Frank, David Basin, and Hitoshi Matsubara. Monte-carlo sampling in games with imperfect information: Empirical investigation and analysis. In *Game Tree Search Workshop*, 1997.

[109] Austin Parker, Dana Nau, and VS Subrahmanian. Game-tree search with combinatorially large belief states. In *International Joint Conference on Artificial Intelligence*, pages 254–259. Professional Book Center, 2005. URL `http://ijcai.org/Proceedings/05/Papers/0878.pdf`.

[110] Austin Parker, Dana S. Nau, and V. S. Subrahmanian. Paranoia versus overconfidence in imperfect information games. In R. Dechter, H. Geffner, and J. Y. Halpern, editors, *Heuristics, Probability and Causality: a Tribute to Judea Pearl*, pages 63–87. College Publications, 2010.

[111] Paolo Ciancarini and Gian Piero Favini. Monte carlo tree search techniques in the game of kriegspiel. In *International Joint Conference on Artificial Intelligence*, volume 9, pages 474–479, 2009. URL `http://ijcai.org/Proceedings/09/Papers/086.pdf`.

[112] Daniel Whitehouse, Edward J Powley, and Peter I Cowling. Determinization and information set monte carlo tree search for the card game dou di zhu. In *Computational Intelligence and Games*, pages 87–94. IEEE, IEEE, 2011. doi: 10.1109/CIG.2011.6031993.

[113] Peter I Cowling, Edward J Powley, and Daniel Whitehouse. Information set monte carlo tree search. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2):120–143, 2012.

[114] Ariel E Bud, David W Albrecht, Ann E Nicholson, and Ingrid Zukerman. Playing "invisible chess" with information-theoretic advisors. In Piotr Gmytrasiewicz and Simon Parsons, editors, *Game Theoretic and Decision Theoretic Agents: Proceedings from the 2001 AAAI Spring Symposium*, pages 6–15. The American Association for Artificial Intelligence, 2001.

[115] Ben George Weber, Michael Mateas, and Arnav Jhala. A particle model for state estimation in real-time strategy games. In *Artificial Intelligence and Interactive Digital Entertainment*, pages 103–108. AAAI Press, 2011. URL `https://www.aaai.org/ocs/index.php/AIIDE/AIIDE11/paper/view/4051`.

[116] Chen Si, Yusuf Pisan, and Chek Tien Tan. A scouting strategy for real-time strategy games. In *Interactive Entertainment*, pages 24:1–24:8, 2014. doi: 10.1145/2677758.2677772.

[117] Kien Quang Nguyen, Zhe Wang, and Ruck Thawonmas. Potential flows for controlling scout units in StarCraft. In *Computational Intelligence and Games*, pages 1–7. IEEE, 2013. doi: 10.1109/CIG.2013.6633662.

[118] Santi Ontañón, Kinshuk Mishra, Neha Sugandh, and Ashwin Ram. On-line case-based planning. *Computational Intelligence*, 26(1):84–119, 2010. doi: 10.1111/j.1467-8640.2009.00344.x.

[119] Ricardo Palma, Antonio A. Sánchez-Ruiz-Granados, Marco Antonio Gómez-Martín, Pedro Pablo Gómez-Martín, and Pedro A. González-Calero. Combining expert knowledge and learning from demonstration in real-time strategy games. In *International Conference on Case-Based Reasoning*, volume 6880 of *Lecture Notes in Computer Science*, pages 181–195. Springer, 2011. doi: 10.1007/978-3-642-23291-6_15.

[120] Alberto Uriarte and Santiago Ontañón. Improving monte carlo tree search policies in StarCraft via probabilistic models learned from replay data. In *Artificial Intelligence and Interactive Digital Entertainment*, pages 100–106. AAAI Press, 2016. URL `https://www.aaai.org/ocs/index.php/AIIDE/AIIDE16/paper/view/13984`.

[121] Matthew Molineaux, David W. Aha, and Philip Moore. Learning continuous action models in a real-time strategy strategy environment. In *International Florida Artificial Intelligence Research Society Conference*, pages 257–262. AAAI Press, 2008. URL `http://www.aaai.org/Library/FLAIRS/2008/flairs08-065.php`.

[122] Hai Hoang, Stephen Lee-Urban, and Héctor Muñoz-Avila. Hierarchical plan representations for encoding strategic game AI. In *Artificial Intelligence and Interactive Digital Entertainment*, pages 63–68. AAAI Press, 2005.

[123] Ben G. Weber, Peter Mawhorter, Michael Mateas, and Arnav Jhala. Reactive planning idioms for multi-scale game AI. In *Computational Intelligence and Games*, pages 115–122. IEEE, 2010. doi: 10.1109/ITW.2010.5593363.

[124] Gabriel Synnaeve and Pierre Bessière. A bayesian tactician. In *Computer Games Workshop at ECAI*, 2012.

[125] Douglas Schneider and Michael Buro. StarCraft unit motion: Analysis and search enhancements. In *AIIDE Workshop on Artificial Intelligence in Adversarial Real-Time Games*, pages 15–21. AAAI Press, 2015. URL `https://www.aaai.org/ocs/index.php/AIIDE/AIIDE15/paper/view/11573`.

[126] Alberto Uriarte and Santiago Ontañón. Game-tree search over high-level game states in RTS games. In *Artificial Intelligence and Interactive Digital Entertainment*. AAAI Press, 2014. URL `https://www.aaai.org/ocs/index.php/AIIDE/AIIDE14/paper/view/8960`.

[127] Dennis Soemers. Tactical planning using MCTS in the game of StarCraft. Master's thesis, Department of Knowledge Engineering, Maastricht University, 2014.

[128] Frederick William Lanchester. *Aircraft in Warfare: The dawn of the Fourth Arm*. Constable and Co., 1916.

[129] James G Taylor. Lanchester-type models of warfare. volume I. Technical report, DTIC Document, 1980.

[130] Marius Stanescu, Nicolas Barriga, and Michael Buro. Using Lanchester attrition laws for combat prediction in StarCraft. In *Artificial Intelligence and Interactive Digital Entertainment*, pages 86–92. AAAI Press, 2015. URL `https://www.aaai.org/ocs/index.php/AIIDE/AIIDE15/paper/view/11531`.

[131] Alberto Uriarte. Multi-reactive planning for real-time strategy games. Master's thesis, Universitat Autònoma de Barcelona, 2011.

[132] Alexander Kovarsky and Michael Buro. Heuristic search applied to abstract combat games. In *Canadian Society for Computational Studies of Intelligence*, volume 3501, pages 66–78. Springer, 2005. doi: 10.1007/11424918_9.

[133] Tung Duc Nguyen, Kien Quang Nguyen, and Ruck Thawonmas. Heuristic search exploiting non-additive and unit properties for RTS-game unit micromanagement. *Journal of Information Processing*, 23(1):2–8, 2015. doi: 10.2197/ipsjjip.23.2.

[134] Gabriel Synnaeve and Pierre Bessière. A dataset for StarCraft AI & an example of armies clustering. In *AIIDE Workshop on Artificial Intelligence in Adversarial Real-Time Games*. AAAI Press, 2012. URL `https://www.aaai.org/ocs/index.php/AIIDE/AIIDE12/paper/view/5513`.

[135] Marius Stanescu, Sergio Poo Hernandez, Graham Erickson, Russel Greiner, and Michael Buro. Predicting army combat outcomes in StarCraft. In *Artificial Intelligence and Interactive Digital Entertainment*. AAAI Press, 2013. URL `https://www.aaai.org/ocs/index.php/AIIDE/AIIDE13/paper/view/7381`.

[136] Antonio A. Sánchez-Ruiz. Predicting the outcome of small battles in StarCraft. In *Workshop Proceedings from The Twenty-Third International Conference on Case-Based Reasoning*, volume 1520 of *CEUR Workshop Proceedings*, pages 33–42. CEUR-WS.org, 2015. URL `http://ceur-ws.org/Vol-1520/paper3.pdf`.

[137] Antonio A Sánchez-Ruiz and Maximiliano Miranda. A machine learning approach to predict the winner in starcraft based on influence maps. *Entertainment Computing*, 19:29–41, 2017.

[138] Peter Emerson. The original Borda count and partial voting. *Social Choice and Welfare*, 40 (2):353–358, 2013.

[139] Alberto Uriarte and Santiago Ontañón. Automatic learning of combat models for RTS games. In *Artificial Intelligence and Interactive Digital Entertainment*, pages 212–219. AAAI Press, 2015. URL `https://www.aaai.org/ocs/index.php/AIIDE/AIIDE15/paper/view/11516`.

[140] Glen Robertson and Ian Watson. An improved dataset and extraction process for StarCraft AI. In *International Florida Artificial Intelligence Research Society Conference*. AAAI Press, 2014.

[141] Olivier Teytaud and Sébastien Flory. Upper confidence trees with short term partial information. In *Applications of Evolutionary Computation*, volume 6624 of *Lecture Notes in Computer Science*, pages 153–162. Springer, 2011. doi: 10.1007/978-3-642-20525-5_16.

[142] David Churchill and Michael Buro. Portfolio greedy search and simulation for large-scale combat in StarCraft. In *Computational Intelligence and Games*, pages 1–8. IEEE, 2013. doi: 10.1109/CIG.2013.6633643.

[143] Marc J. V. Ponsen, Geert Gerritsen, and Guillaume Chaslot. Integrating opponent models with monte-carlo tree search in poker. In *Interactive Decision Theory and Game Theory*, 2010.

[144] Rémi Coulom. Computing Elo ratings of move patterns in the game of Go. *International Computer Games Association*, 30(4):198–208, 2007.

[145] Sylvain Gelly and David Silver. Combining online and offline knowledge in UCT. In *International Conference Machine Learning*, volume 227 of *ACM International Conference Proceeding Series*, pages 273–280. ACM, 2007. doi: 10.1145/1273496.1273531.

[146] Santiago Ontañón. Informed monte carlo tree search for real-time strategy games. In *Computational Intelligence and Games*. IEEE, 2016. doi: 10.1109/CIG.2016.7860394.

[147] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European Conference on Machine Learning*, pages 282–293, 2006. doi: 10.1007/11871842_29.

[148] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002. doi: 10.1023/A:1013689704352.

[149] Christopher D. Rosin. Multi-armed bandits with episode context. In *International Symposium on Artificial Intelligence and Mathematics*, 2010. URL `http://gauss.ececs.uc.edu/Workshops/isaim2010/papers/rosin.pdf`.

[150] Guillaume M JB Chaslot, Mark HM Winands, H JAAP van den Herik, Jos WHM Uiterwijk, and Bruno Bouzy. Progressive strategies for Monte-Carlo tree search. In *Joint Conference on Information Sciences*, pages 655–661, 2007.

[151] Alberto Uriarte and Santiago Ontañón. Combat Models for RTS Games. *IEEE Transactions on Computational Intelligence and AI in Games*, PP(99):1–1, 2017. ISSN 1943-068X. doi: 10.1109/TCIAIG.2017.2669895.

[152] Mark Richards and Eyal Amir. Information set generation in partially observable games. In *AAAI Conference on Artificial Intelligence*. AAAI Press, 2012. URL `http://www.aaai.org/ocs/index.php/AAAI/AAAI12/paper/view/5142`.

[153] Giacomo Bonanno. Memory and perfect recall in extensive games. *Games and Economic Behavior*, 47(2):237–256, 2004.

[154] Alexander Shleyfman, Antonín Komenda, and Carmel Domshlak. On combinatorial actions and cmabs with linear side information. In *European Conference on Artificial Intelligence*, volume 263 of *Frontiers in Artificial Intelligence and Applications*, pages 825–830. IOS Press, 2014.

[155] National Research Council. *Learning to Think Spatially: GIS as a Support System in the K-12 Curriculum*. The National Academies Press, 2006. doi: 10.17226/11019.

[156] Fu Chang, Chun-Jen Chen, and Chi-Jen Lu. A linear-time component-labeling algorithm using contour tracing technique. *Computer Vision and Image Understanding*, 93(2):206–220, 2004. doi: 10.1016/j.cviu.2003.09.002.

[157] David H Douglas and Thomas K Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 10(2):112–122, 1973. doi: 10.3138/fm57-6770-u75u-7727.

[158] S Fortune. A sweepline algorithm for Voronoi diagrams. In *Symposium on Computational Geometry*, pages 313–322. ACM, 1986. doi: 10.1145/10515.10549.

[159] Menelaos I. Karavelas. A robust and efficient implementation for the segment Voronoi diagram. In *Int. Symp. on Voronoi Diagrams in Science and Engineering*, pages 51–62, 2004.

[160] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *International Conference on Management of Data*, pages 47–57. ACM, 1984. doi: 10.1145/602259.602266.

[161] Scott Leutenegger, Mario A. Lopez, and J. Edgington. STR: a simple and efficient algorithm for R-tree packing. In *International Conference on Data Engineering*, pages 497–506, 1997. doi: 10.1109/ICDE.1997.582015.

[162] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Knowledge Discovery and Data Mining*, pages 226–231. AAAI Press, 1996. URL `http://www.aaai.org/Library/KDD/1996/kdd96-037.php`.

[163] Lars Lidén. Strategic and tactical reasoning with waypoints. *AI Game Programming Wisdom*, pages 211–220, 2002.

[164] Alberto Uriarte and Santiago Ontañón. High-level representations for game-tree search in RTS games. In *AIIDE Workshop on Artificial Intelligence in Adversarial Real-Time Games*. AAAI Press, 2014. URL `https://www.aaai.org/ocs/index.php/AIIDE/AIIDE14/paper/view/9015`.

[165] Jack Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965. doi: 10.1147/sj.41.0025.

[166] Matthias Kuchem, Mike Preuss, and Günter Rudolph. Multi-objective assessment of pre-optimized build orders exemplified for StarCraft 2. In *Computational Intelligence and Games*, pages 1–8. IEEE, 2013. doi: 10.1109/CIG.2013.6633626.

[167] Raúl Lara-Cabrera, Carlos Cotta, and Antonio José Fernández Leiva. A self-adaptive evolutionary approach to the evolution of aesthetic maps for a RTS game. In *Congress on Evolutionary Computation*, pages 298–304, 2014. doi: 10.1109/CEC.2014.6900562.

[168] Philippe Codognet and Daniel Diaz. Yet another local search method for constraint solving. In *Stochastic Algorithms: Foundations and Applications*, volume 2264 of *Lecture Notes in Computer Science*, pages 73–90. Springer, 2001. doi: 10.1007/3-540-45322-9_5.

[169] Yves Caniou, Philippe Codognet, Florian Richoux, Daniel Diaz, and Salvador Abreu. Large-scale parallelism for constraint-based local search: the costas array case study. *Constraints*, 20(1):30–56, 2015. doi: 10.1007/s10601-014-9168-4.

[170] Johan Hagelbäck and Stefan J. Johansson. The rise of potential fields in real time strategy bots. In *Artificial Intelligence and Interactive Digital Entertainment*. AAAI Press, 2008. URL `http://www.aaai.org/Library/AIIDE/2008/aiide08-007.php`.

[171] Jay Young and Nick Hawes. Learning micro-management skills in RTS games by imitating experts. In *Artificial Intelligence and Interactive Digital Entertainment*. AAAI Press, 2014. URL `https://www.aaai.org/ocs/index.php/AIIDE/AIIDE14/paper/view/8998`.

[172] Siming Liu, Sushil J. Louis, and Christopher A. Ballinger. Evolving effective micro behaviors in RTS game. In *Computational Intelligence and Games*, pages 1–8. IEEE, 2014. doi: 10.1109/CIG.2014.6932904.

[173] Jay Young, Fran Smith, Christopher Atkinson, Ken Poyner, and Tom Chothia. SCAIL: an integrated StarCraft AI system. In *Computational Intelligence and Games*, pages 438–445. IEEE, 2012. doi: 10.1109/CIG.2012.6374188.

[174] Ricardo Parra and Leonardo Garrido. Bayesian networks for micromanagement decision imitation in the RTS game StarCraft. In *Mexican International Conference on Artificial Intelligence*, volume 7630 of *Lecture Notes in Computer Science*, pages 433–443. Springer, 2012. doi: 10.1007/978-3-642-37798-3_38.

[175] Marcel van der Heijden, Sander Bakkes, and Pieter Spronck. Dynamic formations in real-time strategy games. In *Computational Intelligence and Games*, pages 47–54. IEEE, 2008. doi: 10.1109/CIG.2008.5035620.

[176] Michael Blackadar and Jörg Denzinger. Behavior learning-based testing of StarCraft competition entries. In *Artificial Intelligence and Interactive Digital Entertainment*. AAAI Press, 2011. URL `https://www.aaai.org/ocs/index.php/AIIDE/AIIDE11/paper/view/4045`.

[177] Nicholas Bowen, Jonathan Todd, and Gita Sukthankar. Adjutant bot: An evaluation of unit micromanagement tactics. In *Computational Intelligence and Games*, pages 1–8. IEEE, 2013. doi: 10.1109/CIG.2013.6633664.

[178] Jacky Shunjie Zhen and Ian D. Watson. Neuroevolution for micromanagement in the real-time strategy game StarCraft: Brood War. In *AI 2013: Advances in Artificial Intelligence - 26th Australasian Joint Conference*, volume 8272 of *Lecture Notes in Computer Science*, pages 259–270. Springer, 2013. doi: 10.1007/978-3-319-03680-9_28.

[179] Niels Justesen, Balint Tillman, Julian Togelius, and Sebastian Risi. Script- and cluster-based UCT for StarCraft. In *Computational Intelligence and Games*. IEEE, 2014. doi: 10.1109/CIG.2014.6932900.

Appendix A: StarCraft Concepts

- **Range Unit**: Unit with a weapon range larger than 1 pixel.
- **Melee Unit**: Unit with a weapon range of 1 pixel.
- **Worker**: Unit that can gather resources (minerals and gas).
- **Base**: Building that can store resources.
- **Build order**: List of buildings to build sequentially that defines a strategy.
- **Micro**: Micromanagement of individual unit.
- **Macro**: Control of the economy and production.
- **Technology tree**: in some RTS games, different units or abilities (i.e., "technologies") have to be unlocked by spending resources into "researching" them. Some of these technologies require that the player has first researched some other previous technologies. The list of these technologies that can be researcher, and the dependencies among each other (which ones need to be researched first) is what is known as the technology tree.
- **Fog of war**: a player can only see the areas of the map that her units can reach with their sight.
- **Walling**: strategy consisting in blocking a narrow passage with units.
- **Rush**: strategy consisting in attacking the enemy as soon as possible.
- **Harass**: strategy consisting in attacking workers and ignoring other units.
- **Drop**: strategy consisting in launching a surprise attack with a transport dropping units.

Table A.1: Terran units stats (first group is ground units, second is air units).

	Cost			Attack					
Name	**Gas**	**Mineral**	**HP**	**Ground**	**Air**	**Cool**	**Range**	**Sight**	**Speed**
SCV	50	0	60	5	0	15	1	7	5
Marine	50	0	40	6	6	15	4	7	4
Firebat	50	25	50	16	0	22	2	7	4
Medic	50	25	60	0	0	-	0	9	4
Ghost	25	75	45	10	10	22	7	9	4
Vulture	75	0	80	20	0	30	5	8	6.67
Goliath	100	50	125	12	20	22	5	8	4.7
Siege Tank	150	100	150	70	0	75	12	10	4
Wraith	150	100	120	8	20	30	5	7	6.67
Dropship	100	100	150	0	0	-	0	8	5.47
Valkyrie	250	125	200	0	6	64	6	8	6.6
Science Vessel	100	225	200	0	0	-	0	10	5
Battlecruiser	400	300	500	25	25	30	6	11	2.5

Table A.2: Protoss units stats (first group is ground units, second is air units).

	Cost				Attack					
Name	**Gas**	**Mineral**	**HP**	**Shield**	**Ground**	**Air**	**Cool**	**Range**	**Sight**	**Speed**
Probe	50	0	20	20	5	0	22	1	8	5
Zealot	100	0	100	60	16	0	22	1	7	4
Dragoon	125	50	100	80	20	20	30	4	8	5.25
High Templar	50	150	40	40	0	0	-	0	7	3.33
Dark Templar	125	100	80	40	40	0	30	1	7	5
Archon	100	300	10	350	30	30	20	2	8	5
Dark Archon	250	200	25	200	0	0	-	0	10	5
Reaver	200	100	100	80	100	0	60	8	10	1.77
Shuttle	200	0	80	60	0	0	-	0	8	4.44
Observer	25	75	40	20	0	0	-	0	9	3.33
Corsair	150	100	100	80	0	5	8	5	9	6.67
Scout	275	125	150	100	8	28	30	4	8	5
Carrier	350	250	300	150	6	6	-	8	11	3.33
Arbiter	100	350	200	150	10	10	45	5	9	5

Table A.3: Zerg units stats (first group is ground units, second is air units).

	Cost			Attack					
Name	**Gas**	**Mineral**	**HP**	**Ground**	**Air**	**Cool**	**Range**	**Sight**	**Speed**
Drone	50	0	40	5	0	22	1	7	5
Zergling	25	0	35	5	0	8	1	5	5.57
Hydralisk	75	25	80	10	10	15	4	6	3.71
Lurker	200	200	125	20	0	37	6	8	6
Broodling	0	0	30	4	0	15	1	5	6.04
Infested Terran	100	50	60	500	0	-	1	5	6
Defiler	50	150	80	0	0	-	0	10	4
Ultralisk	200	200	400	20	0	15	1	7	5.4
Overlord	100	0	200	0	0	-	0	9	0.83
Queen	100	100	120	0	0	-	0	10	6.67
Scourge	12	38	25	0	110	-	1	5	6.67
Mutalisk	100	100	120	9	9	30	3	7	6.67
Devourer	250	150	250	0	25	100	6	10	5
Guardian	150	200	150	20	0	30	8	11	2.5

Appendix B: μRTS Concepts

Table B.1: Units stats.

Name	Cost	HP	Attack	Cool	Range	Sight	Speed
Worker	1	1	1	5	1	3	10
Light	2	4	2	5	1	2	8
Heavy	3	8	4	5	1	2	10
Ranged	2	1	1	5	3	3	10

Appendix C: StarCraft AIIDE competition maps analyzed by BWTA2

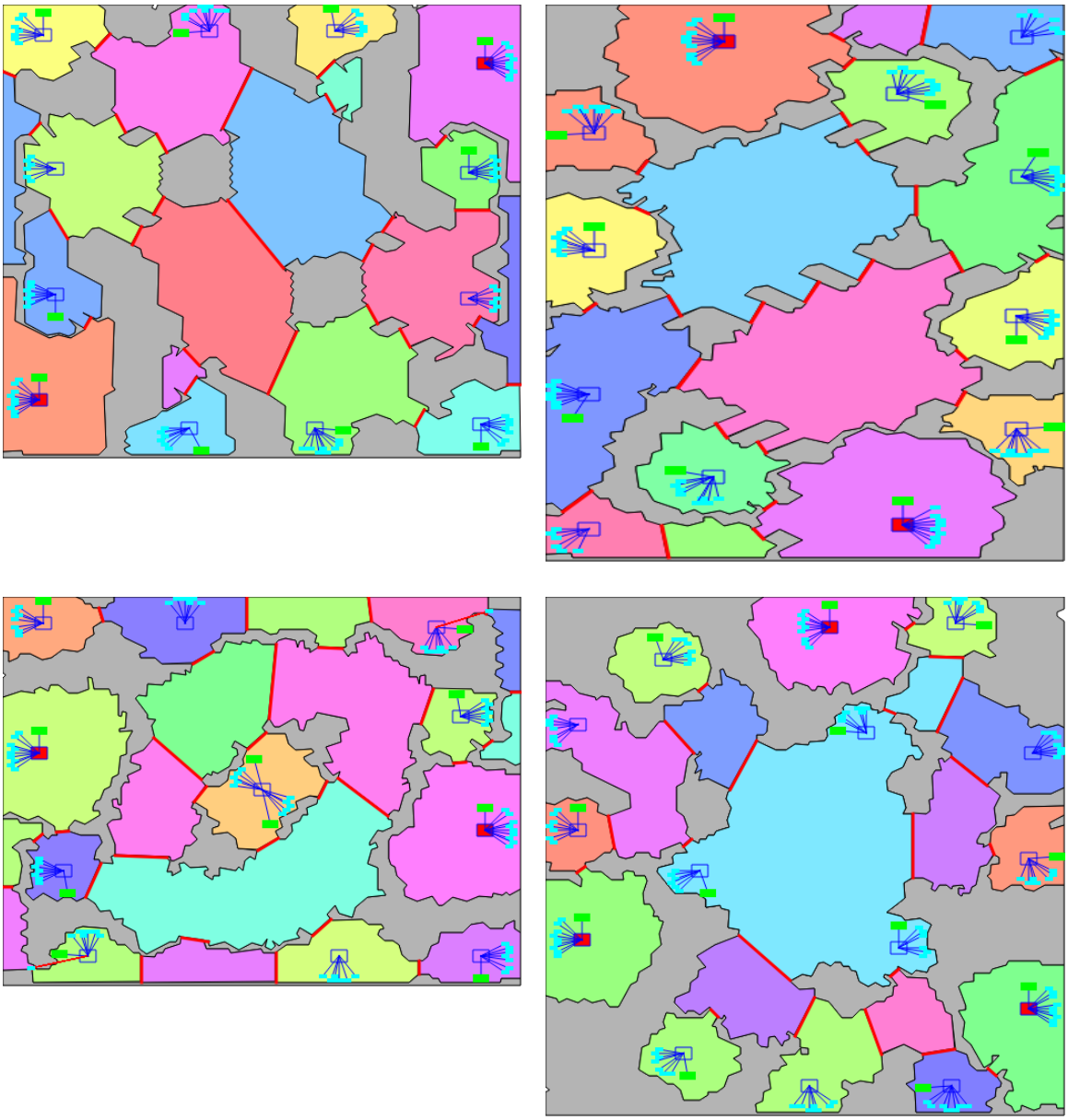


Figure C.1: Map analysis of maps used in StarCraft AIIDE competition.

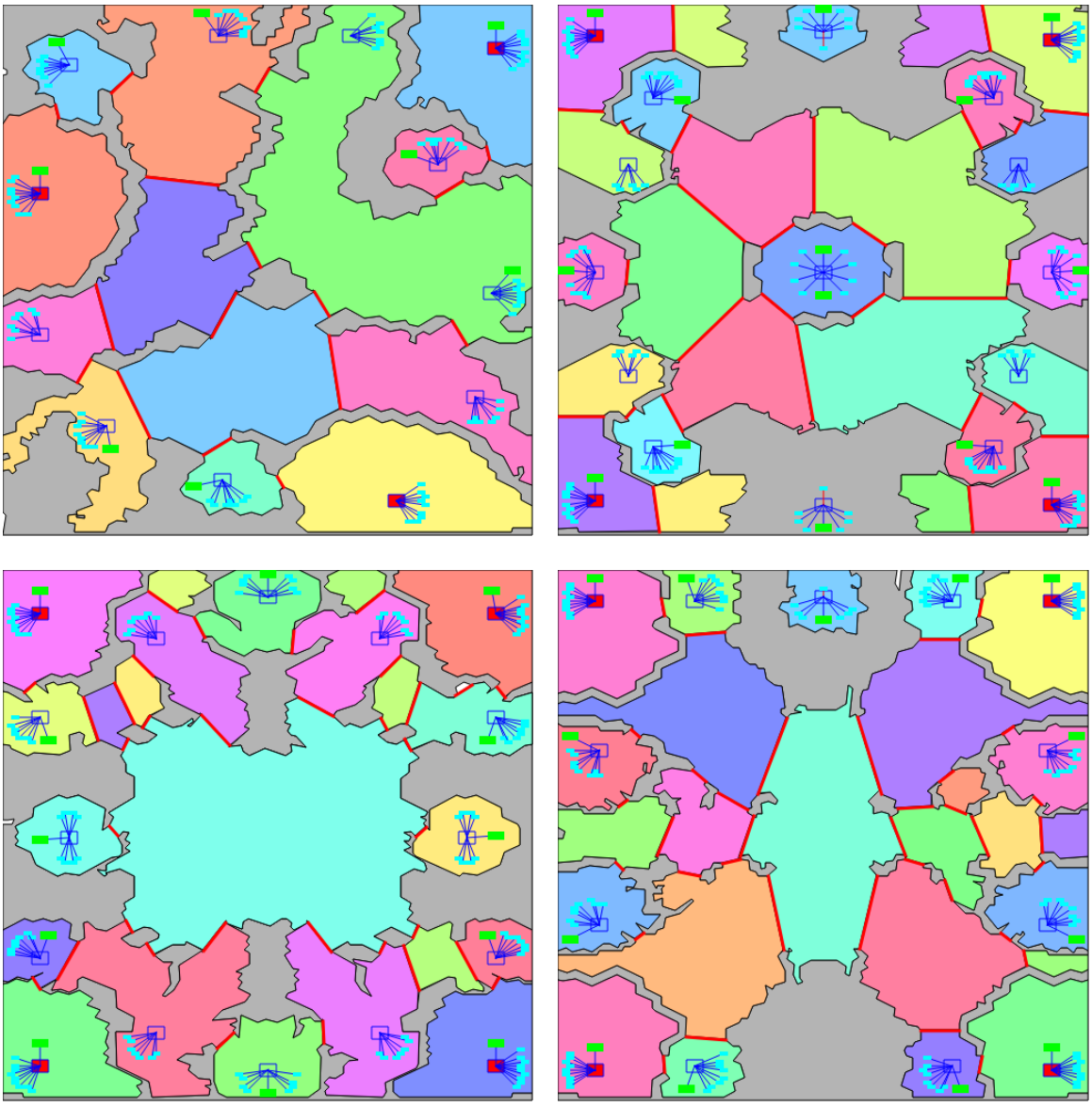


Figure C.2: Map analysis of maps used in StarCraft AIIDE competition.

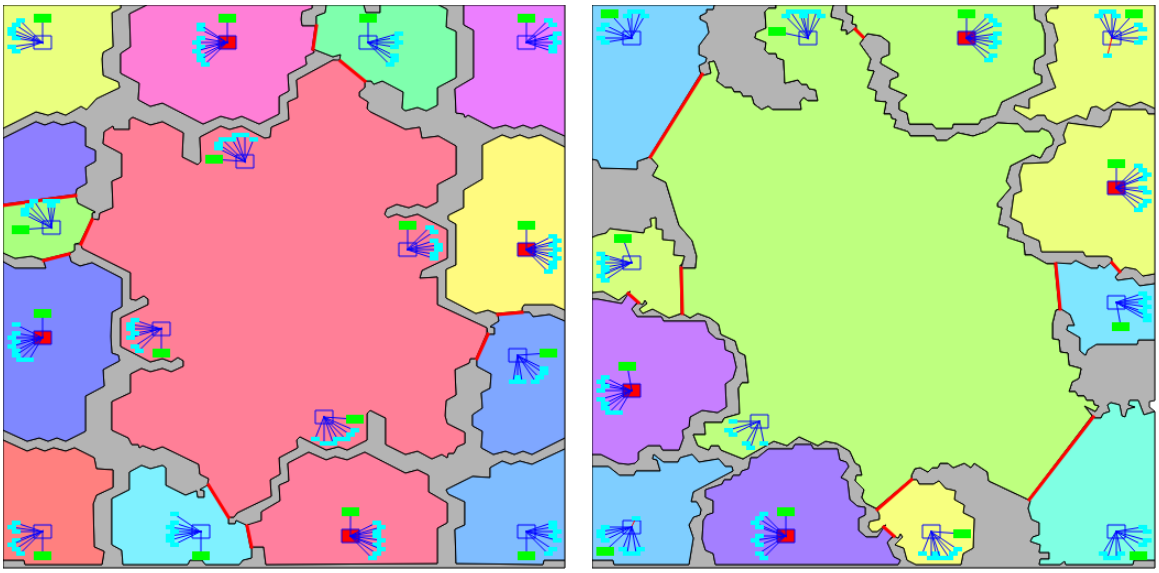


Figure C.3: Map analysis of maps used in StarCraft AIIDE competition.