Tactical Planning Using MCTS in the Game of StarCraft[1]

Dennis Soemers

June 16, 2014

Abstract

This thesis describes how Monte-Carlo Tree Search (MCTS) can be applied to perform tactical planning for an intelligent agent playing full games of *StarCraft: Brood War*. *StarCraft* is a Real-Time Strategy game, which has a large state-space, is played in real-time, and commonly features two opposing players, capable of acting simultaneously. Using the MCTS algorithm for tactical planning is shown to increase the performance of the agent, compared to a scripted approach, when competing on a bot ladder. A combat model, based on Lanchester's Square Law, is described, and shown to achieve another gain in performance when used in Monte-Carlo simulations as replacement for a heuristic linear model. Finally, the MAST enhancement to the Playout Policy of MCTS is described, but it is found not to have a significant impact on the agent's performance.
Keywords: StarCraft, Monte-Carlo Tree Search, Lanchester's Square Law.

1 Introduction

This section first provides background information on the Real-Time Strategy (RTS) game genre, and Artificial Intelligence (AI) research in this game genre. Next, the problem statement and research questions that the thesis aims to address are given. An overview of the remainder of the thesis concludes the section.

1.1 Background

RTS is a genre of games in which players control military units and bases in a real-time environment with the overall goal to destroy all units and buildings controlled by the opponents. Players typically start with a small number of worker units, which are capable of gathering resources and building buildings, and a single building. RTS games are generally played on a 2-dimensional map, which can be observed from a top-down view. Players have to find and gather resources to build bases and armies. The military units in these armies can be controlled individually, and properly doing so helps to win battles.

RTS games have a number of properties that make them notoriously challenging for agents to play, and make it infeasible to use standard AI techniques known from abstract games, such as game-tree search algorithms. Because the games run in *real-time*, decisions have to be made quickly, otherwise the information on which decisions are based may turn out to be outdated. RTS games also typically have two sources of uncertainty. Firstly, unexplored regions of the map are invisible (hidden by the "*Fog of War*"), meaning that players have *imperfect information*. Secondly, combat-related actions can be *non-deterministic* (for instance, shooting units can have a chance to miss their shots). The fact that all players are able to continuously act at the same time, and the fact that many actions are *durative* (meaning that actions take some time to complete), makes it quite difficult to construct a game tree. Finally, RTS games tend to have a state space many orders of magnitude larger than board games such as Chess or Go.

Current approaches to RTS game AI involve decomposing the gameplay in a number of smaller subproblems with different levels of abstraction, and addressing each subproblem individually [1]. One of the subproblems commonly identified is that of tactical decision-making. Tactical decisions, which have to be made in RTS games are, for instance, deciding from which route or angle to attack an opposing army, or where to hide and wait in order to ambush opponents. Tactical decision-making does *not* concern the actions of individual units in a fight (which is commonly referred to as *reactive control* [1]). It also does not involve higher level decisions, such as deciding how many of which unit types to compose an army of. Those kinds of decisions are typically referred to as *strategic reasoning*.

There has been some research into simulation-based approaches to tactical planning in small scenarios based on RTS games. Chung et al. [2] presented a Monte-Carlo planning framework for tactical planning and tested it in a capture-the-flag variant of an RTS game implemented

[1] This thesis has been prepared in partial fulfillment of the requirements for the Degree of Bachelor of Science in Knowledge Engineering, Maastricht University, supervisor: Dr. Mark Winands.

using the ORTS framework [3]. Sailer et al. [4] used a simulation-based planning framework for army deployment in some custom RTS scenarios. Balla and Fern [5] applied the Monte-Carlo Tree Search (MCTS) algorithm [6, 7] to make tactical assault decisions in the RTS game Wargus. Bowen et al. [8] continued their work by using MCTS for tactical groupings in StarCraft. Churchill and Buro [9] applied MCTS to the lower-level control of individual units in combat, but found it to be outperformed by a novel search algorithm named Portfolio Greedy Search.

1.2 Problem Statement and Research Questions

Existing work [5] shows that simulation-based search algorithms, like the MCTS algorithm, work well for tactical planning in isolated RTS scenarios. They have also been shown to be successfully applicable in several board games, such as Go [10]. They have not yet been proven to function well in the context of a full RTS game though, which brings us to the problem statement: "*How can MCTS be applied to successfully handle tactical planning problems in an intelligent agent playing complete RTS games?*" This problem statement involves the following three research questions:

1. How can MCTS be adapted to reason about the durative actions available in an RTS game?
2. How can MCTS perform adequately, with respect to both quality of plans constructed and running time, in a real-time setting?
3. How does MCTS compare to other proposed solutions for tactical planning?

To answer these three questions, an intelligent agent has been developed to play the RTS game *StarCraft: Brood War*. This agent uses a MCTS algorithm for its tactical planning.

1.3 Overview

The thesis is structured as follows. Section 2 provides a description of the game StarCraft: Brood War. The section also describes how the game has been used in the past for RTS game AI research. In Section 3, the overall software architecture of the developed agent is described. Section 4 provides an overview of the techniques used to implement the architecture's modules. Section 5 describes the game state abstraction used to improve the applicability of game tree search algorithms. Section 6 describes the MCTS algorithm and details of how it has been implemented. In Section 7, the setup and results of experiments which were carried out in order to answer the research questions are described. In Section 8, conclusions are drawn from the research and ideas for future research are provided.

2 StarCraft: Brood War

StarCraft: Brood War is a popular and commercially successful RTS game, which was developed by Blizzard Entertainment and released in 1998. A screenshot of the game is shown in Figure 1. It is generally considered to be a remarkably well-balanced considering the complexity of the game [1]. All of the typical properties of RTS games described in the introduction are present in StarCraft. StarCraft is one of the most popular games in academic RTS game AI research. The main reason for that is the Brood War Application Programming Interface (BWAPI).[2] BWAPI is an open source framework that exposes information of the game state to programmers and allows programmers to send commands to the game. This framework allows the programming of agents to play the game in a fast and versatile language like C++. StarCraft is one of, if not the, most complex, successful and well-balanced games for which this option is available.

Figure 1: StarCraft: Brood War

Since 2010, the AIIDE StarCraft AI Competition[3] has been organized every year. Programmers can submit their agents using the BWAPI framework to compete in this competition. At the Computational Intelligence and Games conference (CIG), StarCraft competitions for AI players have been run annually since 2010 as well.[4] In addition to these competitions held at well-known scientific conferences, a number of other competitions have been held too (for instance, the Student StarCraft AI Tournament).[5] The existence of these competitions indicates how popular the game is for academic game AI research.

[2] https://github.com/bwapi/bwapi
[3] http://www.starcraftaicompetition.com
[4] ls11-www.cs.uni-dortmund.de/rts-competition/
[5] http://www.sscaitournament.com/

Before a game starts, each player can choose one of three races to play in that game. Those races are named *Protoss*, *Terran*, and *Zerg*. Each race has different types of units and buildings, and therefore different play styles and strengths and weaknesses. Players start each game with a small number of Worker units and a single Resource Depot. Worker units are units, which can gather resources and construct new buildings. A Resource Depot is a building that can train new Worker units and is required for Worker units to deliver the resources they gather.

In a typical game of StarCraft, players initially focus on gathering resources to quickly expand their economy. They use the resources to build a base and start building an army. Players generally also send one or a couple of units out of the base to scout the opponent. This is done in an attempt to gather information on the opponent's location and his planned strategy. Of course, players can divert from this typical gameplay as they see fit. For example, sometimes players focus more heavily on building a large army quickly to surprise their opponent with an early attack. A game of StarCraft ends once all of the buildings a player constructed throughout the game have been destroyed. The player with no remaining buildings is the losing player.

The number of players that can participate in a single game of StarCraft depends on the map that has been chosen to play the game on, and can be as many as 8 players on the largest maps. However, in the past AI competitions have always focused on 2 players per game (1 versus 1), and therefore this thesis also assumes such a match-up.

3 Agent Architecture

To answer the research questions, an agent has been developed to play full games of StarCraft. The overall software architecture of this agent, named MaasCraft, is shown in Figure 2. The arrows in the figure indicate the flow of data. In some cases it is information about the state of the game, in other cases it can be commands to execute some task. This architecture is largely based on architectures of participating agents in previous competitions, making use of both *Abstraction* and *Divide-and-conquer*, as described in [1].

On the highest level, two large packages can be distinguished; a package for Information and a package for Decision-Making & Control. The arrow from StarCraft to BWAPI indicates BWAPI receiving information on the game state. This information is sent to the relevant modules inside the Information package, where it can be processed and stored. The Decision Making & Control package makes decisions and sends commands to BWAPI, which propagates them to the actual game.

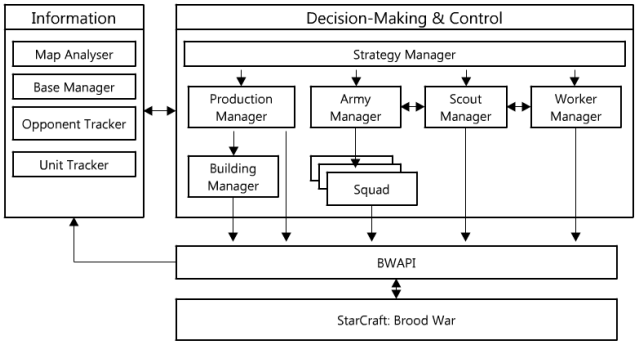

Figure 2: MaasCraft's Software Architecture

3.1 Information

First, the four modules concerning Information are discussed:

- At the beginning of a game, the *Map Analyser* analyses available data about the map that the game is played on. This concerns information that is known to be always the same for a given map. For example, the Map Analyser analyses which of the map's tiles are passable to determine the locations of chokepoints. Throughout the game, this information can be used to make more intelligent decisions.
- The *Base Manager* keeps track of the locations of bases which both players build throughout the game, and which buildings have been observed in each base.
- The *Opponent Tracker* stores a collection of all the opposing units and buildings as they are observed. It also keeps the last known state of each opposing unit of which the agent loses vision in memory.
- The *Unit Tracker* keeps some organizational information on the units controlled by MaasCraft. It ensures that there is only ever a single module which has full control over a given unit. This prevents a single unit getting conflicting commands from multiple different modules.

3.2 Decision-Making & Control

Next, the modules for decision-making & control are discussed. Looking at this part of Figure 2, it should be noted that the higher modules correspond to higher levels of abstraction than the lower modules.

- On the highest level of abstraction is the *Strategy Manager*. This module is responsible for choosing a high-level strategy for the agent to play. This corresponds to the subproblem of strategic reasoning mentioned in Section 1.
- The *Production Manager* considers the decisions of the Strategy Manager with respect to what units

and buildings have to be produced. It is in charge of ordering these production requests. Various production requests are initially ordered by taking into account priority levels indicated by the Strategy Manager. However, the Production Manager can decide to execute lower priority commands first if they require fewer resources. Production commands for training new units are sent straight to BWAPI, since they have no requirements other than resource requirements and the existence of buildings which can train those units. Production commands to construct new buildings are first sent to the Building Manager, since that manager is in charge of finding a location where the building can be placed.

- The *Army Manager* is in charge of making tactical decisions for all of the agent's military units. Generally the military units are grouped in Squads, and the Army Manager makes decisions for these Squads as opposed to individual units.
- The *Scout Manager* is in charge of scouting interesting areas of the map when the Strategy Manager decides to perform scouting. It is linked to both the Army Manager and the Worker Manager since it "borrows" units from one of those managers to use for scouting.
- The *Worker Manager* controls most of the agent's Worker units. Most of these units are gathering resources, and the Worker Manager is in charge of distributing them evenly over the available resources nodes near the agent's bases. When the Scout Manager requests a Worker unit for scouting, the Worker Manager decides which Worker unit is currently best suited for scouting.
- The *Building Manager* receives commands from the Production Manager to construct buildings. It determines a location to construct this building and propagates the command to BWAPI.
- The agent's military units are organized in a number of *Squads*. New Squads can be spawned, or existing Squads can be merged, as the Army Manager sees fit. Each Squad receives a tactical order from the Army Manager, and executes that by sending commands to each of its individual units.

4 Implementation of Modules

This section briefly describes which techniques were used to implement the modules described in Section 3 in MaasCraft. An elaborate description of all the techniques used is outside the scope of this thesis, since the focus lies on the application MCTS to the tactical planning problem. However, since the performance of the agent in a full game partially depends on the performance of all other modules, a short description of their implementation is required.

- The *Map Analyser* uses a number of flood fill algorithms [11] to compute which of the map's tiles are connected to each other, and to compute each tile's clearance. The clearance of a tile is defined to be the straight-line distance from that tile to the nearest blocked tile. Local minima on the clearance map, which satisfy a number of experimentally determined heuristic conditions, are marked as chokepoints. This approach was used by the bot Skynet in a number of previous competitions. Most other bots used a library named Brood War Terrain Analyzer (BWTA)[6], of which the algorithms used are described in [12]. BWTA is no longer compatible with the version of BWAPI used by MaasCraft, which is the main reason for choosing to implement the algorithm used by Skynet.
- The *Strategy Manager* uses a Finite State Machine to select strategies [13]. The states are a number of predefined strategies, and transitions depend on important events in the game. For example, observing a building that produces invisible units causes a transition into a strategy that produces units capable of detecting invisible units. The strategies and the transition conditions are based on expert knowledge.[7]
- The *Production Manager* sorts production commands received from the Strategy Manager, treating the indicated priority levels as preferences (as opposed to strict priorities). It means that the Production Manager can choose to execute lower priority commands first if, for instance, those commands require fewer resources.
- The *Army Manager* uses MCTS for its tactical planning and propagates the chosen actions to their respective Squads. The details of this implementation are described in Sections 5 and 6.
- The *Scout Manager* has a simple rule-based system based on expert knowledge to determine scout locations. Potential fields are used to detect when scouting units are near threatening enemy units and force them to move back to safety.
- The *Worker Manager* uses a number of heuristics based on expert knowledge[7] to spread the Worker units over nearby resource nodes and to determine a good candidate when the Scout Manager requests a unit to use for scouting.

[6] https://code.google.com/p/bwta/
[7] wiki.teamliquid.net/starcraft

- The *Building Manager* performs a breadth-first search to determine the next tile where a building can be legally placed.
- Van der Sterren first described the notion of *Squad* AI for combat games in both a decentralized [14] and a centralized [15] form. MaasCraft uses a centralized approach, where a Squad leader is not an actual member of the Squad, but is the abstract *Army Manager*. Each Squad has access to a set of predefined behaviours and calls the appropriate behaviour on each of its units in order to achieve a goal formulated by the Army Manager. Combat behaviours have been implemented using potential fields [16, 17] for movement, and heuristics based on domain knowledge for target selection [18].

The modules, which have not been discussed, only store data in memory and provide some trivial functionality to access and modify that data.

5 Game State Abstraction

The large size and complexity of the state space of *StarCraft* make it difficult to directly apply game tree search algorithms to the full game. Therefore, an abstraction of the game state is presented in this section, that reduces the complexity of the game state, at the cost of accuracy. The abstract moves, that form the transitions between the states, are also described in this section.

This game state abstraction is intended to be used by some game tree search algorithm. The specific algorithm that is used in MaasCraft is described in the next section.

5.1 Game State

At the start of the game, a graph is built containing nodes at each chokepoint and each potential base location identified by the *Map Analyser*. This spatial abstraction greatly reduces the number of different locations. Assuming the *Map Analyser* functions adequately, this abstraction should still contain all of the important locations. In addition to the graph-representation of the map, which remains constant throughout a game, the abstraction of the game state is defined by the following components:

- A set of allied squads S_A.
- A set of allied bases B_A.
- A set of enemy squads S_E.
- A set of enemy bases B_E.
- A set of battles $\mathcal{B}$.
- A set of scheduled moves $\mathcal{M}$.

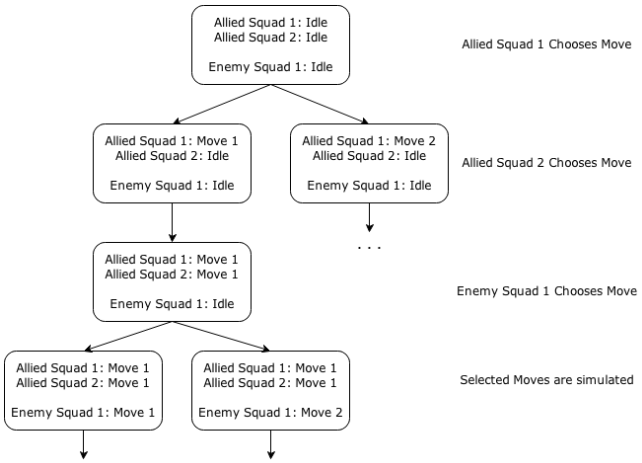

Figure 3: Example Game Tree

Every squad s has a number of hit points ($HP(s)$), representing how much damage it can take before all of its units are dead, a number of damage points it can deal per frame ($DPF(s)$), a maximum movement speed ($Speed(s)$), and a node of the graph on which it is located ($Loc(s)$). These squads represent groups of units in the real game.

Every base b also has a number of hit points ($HP(b)$), and a location ($Loc(b)$), but no speed or damage, because bases are assumed not to be moveable and generally deal no damage. In the game of *StarCraft*, there are some defensive buildings that can deal damage, but these are modelled as separate squads with a speed of 0 at the same location as their bases.

A battle $\beta \in \mathcal{B}$ is characterized by a location $Loc(\beta)$ and a set of participating squads and bases on that same location of which every element must be in $S_A \cup B_A \cup S_E \cup B_E$. At any point in time during a simulation where a node of the graph contains allied squads and/or bases, and enemy squads and/or bases, a battle is constructed at that node and added to $\mathcal{B}$. Bases owned by a specific player are only added as participants to the battle if that player does not have any squads on that node. A battle, as described here, models a local fight between two or more squads, and does *not* describe an entire game. Each game of *StarCraft* generally has many battles.

The purpose of the set of scheduled moves $\mathcal{M}$, is to enable the algorithm to simulate simultaneous moves. It does not only mean that the two players can move simultaneously, but also that a single player can perform multiple moves simultaneously. As proposed in [5] and described in [8], the search tree is built to contain a number of bookkeeping nodes, where moves have been selected but not yet executed. Following an edge in the search tree results in choosing a move for the first idle squad, scheduling that move for execution by adding it

to $\mathcal{M}$, and marking that squad as no longer being idle. The only exception is a node in which every squad has already been assigned an action; such a node is automatically followed up by a special *ResolveMoves* move. This move is not added to $\mathcal{M}$, but advances to the next game state by fully executing the shortest duration move in $\mathcal{M}$, and partially executing all other moves for that same duration. An example game tree for two allied squads and a single enemy squad is depicted in Figure 3.

5.2 Moves

The following moves are available for selection by the squads in the abstract game:

- *Wait(s)* is available to any squad $s \in S_A \cup S_E$ that is not a participant of any battles. It results in s simply waiting where it currently is. This move always has a duration equal to the shortest duration of all other moves in $\mathcal{M}$, which means that squads can re-evaluate whether they still want to continue waiting or not in response to other squads completing their moves.
- *MoveTo(s, n)* is available to any squad $s \in S_A \cup S_E$ that is not a participant of any battles. There is one move of this type for each node n that is adjacent to the current location $Loc(s)$. The duration of this move is given by Equation 1:

$$Duration(MoveTo(s,n)) = \frac{Distance(Loc(s), n)}{Speed(s)} \tag{1}$$

- *Fight(s)* is available to any squad $s \in S_A \cup S_E$ that *is* part of a battle. It indicates that s chooses to continue fighting in this battle. Its duration is equal to the lowest value of either the duration of the battle that s participates in, or the lowest duration of all other moves in $\mathcal{M}$. This means that s can re-evaluate whether it wants to continue fighting in response to other moves, unless all other moves take longer than the entire battle does.
- *RetreatTo(s, n)* is available to any squad $s \in S_A \cup S_E$ that *is* part of a battle. There is one move of this type for each node n that is adjacent to the current location $Loc(s)$ and that is not blocked by an opponent of s. A node n is said to be blocked by an opponent o if o is closer to n than s is. In order to allow opposing squads at the same node to block each other, their positions are not saved as the exact position of a node, but with a slight offset towards the edge they last traversed. This ensures that a squad never attempts to retreat by moving through an opposing squad, but always retreats away from its opponents. The duration is computed in the same way as that of the *MoveTo(s, n)* move (see Equation 1).

5.3 Modelling Battles

Whenever squads end up participating in battles and selecting the *Fight* move, transitioning into the next game state requires the ability to resolve a battle for a specified duration. To determine the duration of *Fight* moves, it is also necessary to predict the maximum duration of each battle. Two different models have been implemented and tested in MaasCraft to perform these two tasks.

In the description of both models, $\mathcal{A}$ and $\mathcal{E}$ denote the sets of all participating allied ($\mathcal{A}$) and enemy ($\mathcal{E}$) squads. The total amount of damage per frame (DPF) and hit points (HP) of each set of squads is given by Equations 2-5. Bases are treated as squads with a DPF value of 0.

$$HP_{Allied} = \sum_{s \in \mathcal{A}} HP(s) \tag{2}$$

$$HP_{Enemy} = \sum_{s \in \mathcal{E}} HP(s) \tag{3}$$

$$DPF_{Allied} = \sum_{s \in \mathcal{A}} DPF(s) \tag{4}$$

$$DPF_{Enemy} = \sum_{s \in \mathcal{E}} DPF(s) \tag{5}$$

Modelling Battles - Heuristic Model

The first model is based on a heuristic used in [19] for comparing the strengths of armies. It assumes that both armies continuously deal their starting amount of DPF to each other, until one of the armies has 0 hit points remaining. In a true battle of *StarCraft* this is not the case, because units die throughout a battle and can no longer deal damage when they are dead, but it is a simple heuristic and computationally inexpensive.

Simulating a battle for a duration of t frames is performed as follows. Let $D = t \times DPF_{Allied}$ denote the total damage all allied squads can deal over a duration of t frames. A battle is then simulated by iterating over all enemy squads s, and subtracting $min(HP(s), D)$ from the squad's hit points and from D, until either all enemy squads have 0 hit points remaining, or $D = 0$. In the same way, enemy squads also deal their damage to allied squads.

The maximum duration of a battle is determined by computing how much time each player requires to kill the opposing participants, and taking the minimum of the two;

$$TimeRequired_{Allied} = \frac{HP_{Enemy}}{DPF_{Allied}} \tag{6}$$

$$TimeRequired_{Enemy} = \frac{HP_{Allied}}{DPF_{Enemy}} \tag{7}$$

Modelling Battles - Lanchester's Square Law

The second model implemented is based on Lanchester's Square Law [20]. Lanchester's Square Law models a battle using differential equations, that describe how the number of units in each army changes over time. It is intended to model modern combat with aimed fire, as opposed to Lanchester's Linear Law, that models classic hand-to-hand combat. See Equation 8, where $m(t)$ denotes the number of allied units at time t, and $n(t)$ denotes the number of enemy units at time t.

$$\frac{dm}{dt} = -an, \frac{dn}{dt} = -bm \tag{8}$$

In this model, a indicates how many allied units each enemy unit is expected to kill per time unit, and b is the analogous variable for enemy units. This model is named the square law, because doubling the initial number of units in an army has the same effect on the end result of the battle, as quadrupling the fighting effectiveness (a or b) of each unit in that army.

An extension to the model, that models reinforcements arriving during a battle, was described in [21]. Adding this extension results in the differential equations shown in Equation 9, where P and Q are constants representing the number of extra units arriving to the battle for the allied and enemy armies, respectively.

$$\frac{dm}{dt} = P - an, \frac{dn}{dt} = Q - bm \tag{9}$$

The model described in Equation 9 has been implemented and tested in MaasCraft. The parameters a and b are computed as shown in Equations 10 and 11. In these equations, $\overline{DPF}(S)$ and $\overline{HP}(S)$ denote the mean damage per frame and hit points, respectively, of all units in the squads in S.

$$a = \frac{\overline{DPF}(\mathcal{E})}{\overline{HP}(\mathcal{A})} \tag{10}$$

$$b = \frac{\overline{DPF}(\mathcal{A})}{\overline{HP}(\mathcal{E})} \tag{11}$$

The variables P and Q are set to $\frac{1}{Z}$ if the respective army has a base located at the battle location, and 0 otherwise, where Z equals the time in frames it takes to train a Zealot[8] in *StarCraft*. Given a point in time t, Equations 14 and 15 provide a closed-form solution for $m(t)$ and $n(t)$. Equations 12 and 13 give intermediate results used in Equations 14 and 15.

$$E = \frac{1}{2}\left\{\left[m_0 + \frac{P}{\sqrt{ab}}\right] - \frac{\sqrt{ab}}{b}\left[n_0 + \frac{Q}{\sqrt{ab}}\right]\right\} \tag{12}$$

[8]The most basic unit type of the Protoss race.

$$F = \frac{1}{2}\left\{\frac{\sqrt{ab}}{a}\left[m_0 + \frac{P}{\sqrt{ab}}\right] + \left[n_0 + \frac{Q}{\sqrt{ab}}\right]\right\} \tag{13}$$

$$m(t) = \frac{Q}{b} + Ee^{t\sqrt{ab}} + \frac{\sqrt{ab}}{b}Fe^{-t\sqrt{ab}} \tag{14}$$

$$n(t) = \frac{P}{a} - \frac{\sqrt{ab}}{a}Ee^{t\sqrt{ab}} + Fe^{-t\sqrt{ab}} \tag{15}$$

The variable E computed in Equation 12, which is only dependent on the model's initial conditions, can be used to predict ahead of time which army wins the battle. The allied army wins if $E > 0$, and the enemy army wins if $E < 0$. If $E = 0.0$, both armies die simultaneously according to the model.

A battle is simulated for a duration of t frames by computing the number of units remaining in each army using Equations 14 and 15. Then, the remaining total HP and DPF for each squad S is re-computed by multiplying the remaining number of units by $\overline{HP}(S)$ and $\overline{DPF}(S)$, respectively. This allows those squads to participate in future battles in an appropriately damaged state.

The maximum duration of a battle can be computed as shown in Equation 18, where Equations 16 and 17 provide intermediate results dependent on the sign of E.

$$D = \begin{cases} P^2 + 4aEF\sqrt{ab}, & E > 0 \\ Q^2 - 4bEF\sqrt{ab}, & E < 0 \end{cases} \tag{16}$$

$$z = \begin{cases} \frac{-P+\sqrt{D}}{2aF}, & E > 0 \\ \frac{-Q+\sqrt{D}}{2F\sqrt{ab}}, & E < 0 \end{cases} \tag{17}$$

$$t_{end} = \frac{-ln(z)}{\sqrt{ab}} \tag{18}$$

In the rare cases where $E = 0$, t_{end} is directly computed using $z = 0.001$. This value was found to be accurate for some cases. Whether it is possible at all to find a constant value or a function for z, that gives accurate results for t_{end} in all cases where $E = 0$, has not been investigated for this thesis.

6 MCTS for Tactical Planning

This section first describes Monte-Carlo Tree Search (MCTS) in general. This is followed by a description of the UCT algorithm, which is a specific variant of MCTS that has been implemented in MaasCraft. Finally, the section describes how the algorithm has been implemented in MaasCraft and how it is used by the *Army Manager*.

6.1 Monte-Carlo Tree Search

MCTS is a family of best-first search algorithms [6, 7]. MCTS uses Monte-Carlo simulations to gradually build a tree over the state-space. Every simulation results in a path from the current game state (root node) to some terminal state. Statistics of previous simulations are used to guide the search towards the most interesting parts of the tree. Typically, MCTS consists of four different steps which are repeated for as long as a given computational budget (such as a time constraint) allows [22]. These four steps are depicted in Figure 4.

In the *(1) Selection Step*, the tree is traversed from the root node to a leaf node. In this step, exploration of new moves, and exploitation of moves which seem successful so far, should be balanced. In the *(2) Expansion Step*, one or more children are added to the leaf node. In the *(3) Simulation Step*, a game is simulated from the leaf node up until a terminal state is reached. Finally, in the *(4) Back-propagation Step*, a reward value corresponding to the terminal state that was reached is back-propagated through the nodes along the search path.

The selection and expansion steps can be grouped in a *Tree Policy*. The policy used for the simulation step is often referred to as the *Playout Policy*.

6.2 UCT

In MaasCraft, the UCT algorithm [6] has been implemented following the pseudocode in [10]. This variant of the algorithm makes no assumptions with respect to the turn order, which is important because of the real-time and simultaneous-move nature of the problem. The policies of MCTS mentioned above are implemented as follows in UCT:

- **Tree Policy:** A node is said to be *fully expanded* if and only if all of its potential children have been added to the tree. If the current node has not been fully expanded yet, an arbitrary child, which has not been added yet, is selected and added to the tree. Otherwise, the child node v' of the current node v that maximizes Equation 19 (based on the UCB1 policy [23]), where $R(n)$ is the average reward value found for the node n so far, and $N(n)$ is the number of times a node n has been visited.

$$X(v') = R(v') + C\sqrt{\frac{ln(N(v))}{N(v')}} \tag{19}$$

  This formula balances exploration of nodes that have not been visited often with exploitation of nodes that seem rewarding. C is a parameter that can be chosen to control this balance. Higher values for C result in a larger amount of time spent exploring (as opposed to exploiting). The exact way that $R(v')$ is computed in MaasCraft is an implementation detail and therefore described in Subsection 6.4.

- **Playout Policy:** The most straightforward policy for simulation, is to pick a random action from a uniform distribution over the possible actions in each game state. This is the policy implemented in all but one versions of MaasCraft. An enhancement, that has been implemented in one version of MaasCraft, is the Move-Average Sampling Technique (MAST) [24]. This technique saves a value $Q_h(a)$ for every move a, which is the average of all the rewards obtained in play-outs in which move a occurred. The idea is that moves that were good in one state are more likely to also be good in other states, and should therefore be chosen more often. As proposed in [25], the ϵ-greedy [26] technique is used to select the move with the highest $Q_h(a)$ value with probability $1-\epsilon$, and to randomly select a move from a uniform distribution with probability ϵ. In MaasCraft, $\epsilon = 0.6$. This value was determined by trial and error.

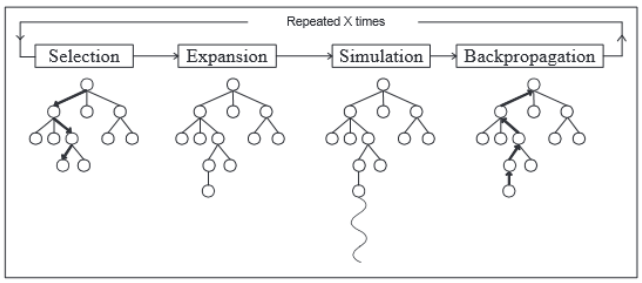

Figure 4: The four steps of MCTS [22]

6.3 Implementation

Terminal States

In the implementation in MaasCraft, a game state is said to be terminal if at least one of the following three conditions holds:

1. 1440 or more frames of time have been simulated. At 24 frames per second, which is the highest speed available by default in *StarCraft*, this corresponds to 1 minute. This means that the algorithm plans at most 1 minute into the future.
2. All squads have selected the *Wait* move. According to the rules described in Section 5, none of the squads are able to re-evaluate their moves at this point, so the simulation can be terminated.
3. All squads and bases of a single player have 0 hit points remaining. When this happens one of the players has lost all of its forces and there is no point in continuing the simulation.

The reward value R for a terminal game state is computed according to Equation 23, consisting of a number of components computed in Equations 20-22.

$$BaseScore = \Delta |B_A| - \Delta |B_E| \quad (20)$$

where Δ denotes the change in the quantity from the start state to the terminal state.

$$HpScore = \frac{TotalHP^{end}_{Allied}}{TotalHP^{start}_{Allied}} - \frac{TotalHP^{end}_{Enemy}}{TotalHP^{start}_{Enemy}} \quad (21)$$

$$DistanceScore = \epsilon \sum_{s \in S_A} \left[\sum_{b \in B_A} Distance(s,b) - \sum_{b \in B_E} Distance(s,b) \right] \quad (22)$$

$$R = BaseScore + HpScore + DistanceScore \quad (23)$$

The *BaseScore* computed in Equation 20 gives a reward of 1 for every enemy base destroyed, and deducts 1 for every allied base lost. This means that this component of the score lies in $[-m, n]$, where m and n denote the number of allied and enemy bases, respectively, at the start of the simulation.

The *HpScore* computed in Equation 21 gives a reward in $[-1, 1]$ depending on the total hit points the armies of both players have remaining.

The *DistanceScore* computed in Equation 22 rewards squads for moving away from allied bases and towards opposing bases. The parameter ϵ should be chosen to be small (in MaasCraft, $\epsilon = 10^{-6}$), to ensure that the *BaseScore* and *HpScore* is given a larger weight. The *DistanceScore* is, in essence, only used as a tie-breaker for simulations where no battles occurred. Because MaasCraft, as of the time of this writing, only has a small selection of early aggressive strategies implemented, it is generally beneficial to move aggressively towards the enemy bases.

UCT's C parameter

The C parameter used by UCT (see Equation 19) is set to a constant value of $1.5\sqrt{2}$, chosen by hand-tuning. The C parameter should be correlated to the range in which a final state's reward value R (Equation 23) can lie. This range is dominated by the *BaseScore* value, which contributes -1 to the lower bound for each allied base and 1 to the upper bound for each enemy base, and the *HpScore*, which lies in $[-1, 1]$.

Ideally, the C parameter should scale with the range in which the reward values can lie, or the reward values should be normalized to a constant range. Considering MaasCraft's current focus on early-game strategies, most games in which the bot participates have a relatively small number of bases, which means that the range of reward values ends up fairly small and consistent in practice. For this reason, no effort has been put into perfecting the value of the C parameter yet.

Interface to Army Manager

MaasCraft's *Army Manager* initializes a new UCT search every 15 frames. Whenever a new search is initialized, an abstraction of the game state is constructed at the root node to represent the true game state as accurately as possible. To do so, the DBSCAN clustering algorithm [27] is used to cluster units into squads based on proximity (the specific choice of clustering algorithm should not matter much). Allied squads require at least two units close to each other; loose units that are not assigned to squads are not considered by UCT, but are instead directly instructed to move towards the closest squad. This is done because a single unit in StarCraft is rarely able to accomplish much on its own. Enemy squads *are* allowed to consist of only a single unit though. This allows the algorithm to recognize when the opponent left a single unit vulnerable and exploit the situation by attacking with a larger squad.

During each frame, the algorithm is allowed to run for 30 milliseconds. This leaves sufficient time available per frame for all the other modules to perform their tasks and ensure the agent does not exceed the limit of 42 milliseconds per frame enforced by the competition rules of the AIIDE and CIG competitions.

New moves based on the search results are only selected and executed at the end of each round of 15 frames, right before a new search is initialized. It means that the selected moves in the true game are based on slightly outdated information, and the agent requires a period of at least 15 frames to change its tactics based on new information. Because the algorithm is only used for larger scale tactical decisions (as opposed to, for instance, directly controlling individual units in combat), this turns out to be acceptable in practice.

The output expected from the algorithm is a *Wait* or *Move* command for every allied squad that is not currently in combat, and a *Fight* or *Retreat* command for every allied squad that *is* currently in combat. This results in the *Army Manager* moving groups of units around the map, attempting to maximize the score given by Equation 23 it expects to be capable of achieving 1 minute into the future.

7 Experiments

7.1 Setup

A number of different versions of MaasCraft have been submitted to the StarCraft BroodWar Bots Lad-

der.[9] This is a website that is automatized to continuously run games between BWAPI-based bots. Most of the bots participating on the ladder have also participated in competitions, such as the ones at AIIDE and CIG, in previous years, but there is also a number of bots that have not been seen in previous competitions.

The website ranks bots according to an ELO rating system [28]. The ELO rating of a bot gives an indication of the bot's relative performance, compared to the other participants. The website also provides access to other statistics, such as total win, loss, and crash percentages.

During development of the bot, a total of 12 different versions of MaasCraft have been submitted to the ladder. Of four of these versions, the results are presented in this section. The other versions only played a small number of games, and served only for quick testing and identifying bugs and other issues. The four versions that played larger numbers of games and are compared in this section are:

1. MaasCraft_0_3: This is the last version that was submitted before development of the MCTS algorithm for tactical reasoning started. Instead of MCTS, it uses a simple scripted approach, where the complete armies of each player are compared directly using a heuristic. Based on this comparison the bot either attacks or retreats with its entire army.
2. MaasCraft_0_7: This version uses the MCTS algorithm for its tactical planning as described in Section 5. It uses the Heuristic Model for simulating battles in the Tree and Playout policies of MCTS.
3. MaasCraft_0_10: This version uses the MCTS algorithm for its tactical planning as described in Section 5. It uses the model based on Lanchester's Square Law for simulating battles in the Tree and Playout policies of MCTS.
4. MaasCraft_0_12: This version is similar to MaasCraft_0_10, but uses the MAST enhancement in its playout policy, as described in Section 6.

Additionally, the same four versions have played a number of games against the Racine AI (version 3.0.1), which is a fully scripted AI that can be installed to replace the built-in AI of *StarCraft*. Compared to BWAPI-based bots, this AI has less control over individual units and therefore, in general, has a lower performance in equal fights. However, it has perfect information and gains free resources, meaning it generally has a stronger army than BWAPI-based bots, which do not cheat. This script provides a more consistent benchmark than the bot ladder, on which other bots can get enabled, disabled and upgraded in between experiments.

[9] http://bots-stats.krasi0.com/

Table 1: MaasCraft Game Statistics

Version	Win %	Loss %	Draw %	Games
0_3	16.07%	76.43%	7.50%	280
0_7	30.22%	67.03%	2.75%	182
0_10	40.61%	56.85%	2.54%	197
0_12	46.41%	51.93%	1.66%	181

Table 2: MaasCraft ELO Ratings

Version	ELO Peak	Final ELO	Date
0_3	1867	1783	22-05-2014
0_7	2009	1894	31-05-2014
0_10	2153	2028	05-06-2014
0_12	2116	2099	14-06-2014

Using Racine, each of the four versions of MaasCraft described above played a set of 4 games versus each of the game's 3 available races, on each of the 10 maps used in the AIIDE 2013 competition, for a total of 120 games per version of MaasCraft.

7.2 Results

Table 1 shows the win, loss and draw percentages and the total number of games played for each version of MaasCraft. Games in which MaasCraft crashed are not included in these statistics. In a tournament setting, crashes would be treated as losses and all versions of the bot would have lower win percentages than presented in Table 1. For the purpose of this thesis, however, it is assumed that crashes are not related to the approaches being compared, and therefore excluded from the experiments. Draws occur on the bot ladder when, after 1 hour of gameplay, neither player has won. Generally, when this occurs, one of the players has essentially won the game, but has not been programmed to scout the full map for any remaining opposing buildings to properly finish the game.

Overall, Table 1 shows that MCTS gives a performance improvement, with respect to win and loss percentages, over the scripted approach to tactical reasoning. The combat model based on Lanchester's Square Law improves the performance of MCTS even more. Adding MAST to the Playout Policy of MCTS seems to increase the win percentage. Because other bots on the ladder can get disabled, enabled, or updated, the ladder is not a static environment, and such changes can also affect the percentages. The ELO ratings shown in Table 2 are less affected by these changes in the test environment. In Table 2, the peak ELO is the highest rating that a version obtained throughout its time on the ladder. The final ELO is the ELO rating that the bot

Table 3: MaasCraft Win Percentages Per Map

Version	(3)Tau Cr.	(4)Andro.	(4)Pyth.
0_3	21.36%	21.67%	10.42%
0_7	34.00%	29.58%	30.36%
0_10	52.54%	37.18%	36.36%
0_12	54.76%	42.42%	42.42%

Table 4: MaasCraft Win Percentages Vs. Racine

Version	Vs. Protoss	Vs. Terran	Vs. Zerg
0_3	0.0%	0.0%	0.0%
0_7	12.5 ± 10.25%	32.5 ± 14.52%	0.0%
0_10	15.0 ± 11.07%	40.0 ± 15.18%	0.0%
0_12	15.0 ± 11.07%	45.0 ± 15.42%	0.0%

had on the date in the last column. This table shows similar changes to Table 1 for the first three versions of the bot. It shows that adding MAST in the last version does not have a significant impact. Table 3 shows the win percentages that MaasCraft achieved on the three different maps that were used on the bot ladder (*Tau Cross*, *Andromeda* and *Python*). The number in front of a map name indicates the number of players that can theoretically play on that map, and is generally an indicator of map size. On maps for fewer players, it is easier to scout and early-game aggressive strategies are more likely to succeed. This is indeed reflected in Table 3, where the 3-player map is the only map on which versions of MaasCraft achieved win percentages over 50%.

Finally, Table 4 shows the win percentages achieved by all four versions of MaasCraft, against the three different races controlled by the Racine script, and the corresponding 95% confidence intervals. A number of observations can be made in this table. It shows that none of the versions of MaasCraft were every able to beat the Racine script when it controlled the *Zerg* race. It is likely that the specific strategy that the Racine script uses when playing *Zerg* is particularly strong against MaasCraft's strategy.

When playing against the *Protoss* and *Terran* races, there is a large improvement in performance adding MCTS, going from version 0_3 to 0_7. Against the *Protoss* race, version upgrades other than adding the initial MCTS algorithm did not make a significant difference. Against the *Terran* race, the other version upgrades seem to have a larger influence, but still require a larger sample size before this can be definitively concluded.

8 Conclusions & Future Research

In this thesis, the MCTS algorithm has been shown to be applicable to the tactical planning problem of an intelligent agent playing the game of *StarCraft*. An abstraction of the game state has been presented that reduces the complexity of the problem enough for the algorithm to perform better than a scripted approach, even under harsh time-constraints. In order to deal with durative and simultaneous moves, the game tree contains bookkeeping nodes that only memorize which moves have been selected but do not advance the game state further.

The algorithm has been shown to achieve a gain in performance, measured by win percentage and ELO rating, compared to a simple scripted approach. Additionally, an improved combat model based on Lanchester's Square Law has been shown to increase the performance of the MCTS algorithm further, when applied in the algorithm's Tree and Playout policies. No conclusive empirical evidence could be found whether MAST improves the playout, though the trend suggests it might have a positive effect.

It is difficult to judge the performance of MCTS for tactical reasoning with respect to other proposed solutions. Most of the effort during development of MaasCraft has been spent on the MCTS algorithm for tactical reasoning, meaning that other aspects of the bot, such as its strategic reasoning, are likely to be weaker than those of other bots on the bot ladder and in competitions. This has to be addressed, before a fair comparison of the tactical reasoning capabilities can be made with the strongest bots.

Ideas for future research that address these areas can be found in [1, 29], where an overview of existing work and open questions in RTS game AI can be found. The MCTS algorithm itself can still be improved as well. Some of the time management strategies described in [30] can be applied to reduce the time used by MCTS if enhancements to other aspects of the bot require more time per frame. A decay factor can be applied to reuse the search tree continuously [31], as opposed to reconstructing a completely new search after a set number of frames. The combat model based on Lanchester's Square Law can be extended, for instance to better model the effect of different weapon ranges or heterogeneous army compositions [32, 33]. Many proposed extensions for the model remove the ability of deriving an analytical solution though, which makes the use of numerical methods necessary, and influences the simulation speed.

References

[1] Ontañón, S., Synnaeve, G., Uriarte, A., Richoux, F., Churchill, D., and Preuss, M. (2013). A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft. *Computation Intelligence and AI in Games, IEEE Transactions on*, Vol. 5, No. 4, pp. 293–311.

[2] Chung, M., Buro, M., and Schaeffer, J. (2005). Monte Carlo Planning in RTS Games. *IEEE Symposium on Computational Intelligence and Games (CIG)* (eds. G. Kendall and

S. Lucas), pp. 117–124, IEEE Computer Society Press, Los Alamitos.

[3] Buro, M. (2003). ORTS A Hack-Free RTS Game Environment. *Proceedings of the Third International Conference on Computers and Games* (eds. J. Schaeffer, M. Müller, and Y. Björnsson), Vol. 2883 of *Lecture Notes in Computer Science*, pp. 280–291, Springer Berlin Heidelberg.

[4] Sailer, F., Buro, M., and Lanctot, M. (2007). Adversarial Planning Through Strategy Simulation. *IEEE Symposium on Computational Intelligence and Games (CIG)*, pp. 80–87.

[5] Balla, R.-K. and Fern, A. (2009). UCT for Tactical Assault Planning in Real-Time Strategy Games. *International Join Conference of Artifical Intelligence, IJCAI*, pp. 40–45, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[6] Kocsis, L. and Szepesvári, C. (2006). Bandit Based Monte-Carlo Planning. *Proceedings of the 17th European Conference on Machine Learning (ECML 2006)* (eds. J. Fürnkranz, T. Scheffer, and M. Spiliopoulou), Vol. 4212 of *Lecture Notes in Computer Science*, pp. 282–293, Springer Berlin Heidelberg.

[7] Coulom, R. (2007). Efficient Selectivity and Backup Operators in Monte-Carlo tree search. *Computers and Games* (eds. H. J. van den Herik, P. Ciancarini, and H. H. L. M. Donkers), Vol. 4630 of *Lecture Notes in Computer Science*, pp. 72–83, Springer Berlin Heidelberg.

[8] Bowen, N., Todd, J., and Sukthankar, G. (2013). Adjutant Bot: An Evaluation of Unit Micromanagement Tactics. *Proceedings of the IEEE Conference on Computational Intelligence in Games*, pp. 336–343, Niagara Falls, Canada.

[9] Churchill, D. and Buro, M. (2013). Portfolio Greedy Search and Simulation for Large-Scale Combat in StarCraft. *Proceedings of the IEEE Conference on Computational Intelligence in Games*, pp. 217–224.

[10] Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. (2012). A Survey of Monte Carlo Tree Search Methods. *Computation Intelligence and AI in Games, IEEE Transactions on*, Vol. 4, No. 1, pp. 1–43.

[11] Pavlidis, T. (1982). *Algorithms for Computer Graphics and Image Processing*. Computer Science Press, Potomac, MD.

[12] Perkins, L. (2010). Terrain Analysis in Real-Time Strategy Games: An Integrated Approach to Choke Point Detection and Region Decomposition. *Proceedings of the Sixth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2010* (eds. G. M. Youngblood and V. Bulitko), pp. 168–173, The AAAI Press.

[13] Millington, I. (2006). *Artificial Intelligence for Games*. CRC Press.

[14] Sterren, W. van der (2002a). Squad Tactics: Team AI and Emergent Maneuvers. *AI Game Programming Wisdom* (ed. S. Rabin), Chapter 5.3, pp. 233–246. Charles River Media.

[15] Sterren, W. van der (2002b). Squad Tactics: Planned Maneuvers. *AI Game Programming Wisdom* (ed. S. Rabin), Chapter 5.4, pp. 247–259. Charles River Media.

[16] Hagelbäck, J. and Johansson, S. J. (2008). The Rise of Potential Fields in Real Time Strategy Bots. *Proceedings of AIIDE 2008* (eds. C. Darken and M. Mateas), pp. 42–47.

[17] Hagelbäck, J. and Johansson, S. J. (2009). A Multi-agent Potential Field-Based Bot for Real-Time Strategy Games. *International Journal of Computer Games Technology*, Vol. 2009.

[18] Uriarte, A. and Ontañón, S. (2012). Kiting in RTS Games Using Influence Maps. *AIIDE Workshop on Artificial Intelligence in Adversarial Real-Time Games*, pp. 31–36.

[19] Uriarte, A. (2011). Multi-Reactive Planning for Real-Time Strategy Games. M.Sc. thesis, Universitat Autònoma de Barcelona, Spain.

[20] Lanchester, F. W. (1916). *Aircraft in Warfare: The Dawn of the Fourth Arm*. Constable Limited.

[21] Morse, P. M. and Kimball, G. E. (1946). Methods of Operations Research. Technical Report OEG–54, Center for Naval Analyses Operations Evaluation Group, Alexandria, VA.

[22] Chaslot, G. M. J.-B., Winands, M. H. M., Herik, H. J. van den, Uiterwijk, J. W. H. M., and Bouzy, B. (2008). Progressive Strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation*, Vol. 4, No. 3, pp. 343–357.

[23] Auer, P., Cesa-Bianchi, N., and Fischer, P. (2002). Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, Vol. 47, No. 2-3, pp. 235–256.

[24] Finnsson, H. and Björnsson, Y. (2008). Simulation-Based Approach to General Game Playing. *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, pp. 259–264.

[25] Tak, M. J. W., Winands, M. H. M., and Björnsson, Y. (2012). N-Grams and the Last-Good-Reply Policy applied in General Game Playing. *Computational Intelligence and AI in Games, IEEE Transactions on*, Vol. 4, No. 2, pp. 73–83.

[26] Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning. The MIT Press, Cambridge, Massachusetts.

[27] Ester, M., Kriegel, H.-P., Sander, J., and Xu, X. (1996). A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, Vol. 96, pp. 226–231, AAAI Press.

[28] Elo, A. E. (1978). *The Rating of Chess Players: Past and Present*. Arco Publishing, New York.

[29] Roberston, G. and Watson, I. (2014). A Review of Real-Time Strategy Game AI. *AI Magazine*. In Press.

[30] Baier, H. and Winands, M. H. M. (2012). Time Management for Monte-Carlo Tree Search in Go. *Advances in Computer Games* (eds. H. J. van den Herik and A. Plaat), Vol. 7168 of *Lecture Notes in Computer Science*, pp. 39–51. Springer Berlin Heidelberg.

[31] Pepels, T., Winands, M. H. M., and Lanctot, M. (2014). Real-Time Monte-Carlo Tree Search in Ms Pac-Man. *Computational Intelligence and AI in Games, IEEE Transactions on*. In Press.

[32] Bonder, S. (1965). A Theory for Weapon System Analysis. *Proceedings U.S. Army Operations Research Symposium*, Vol. 4, pp. 111–128.

[33] Bonder, S. and Farrell, R. (1970). Development of Models for Defense Systems Planning. Technical Report SRL–2147–TR–70–2, Systems Research Laboratory, The University of Michigan, Ann Arbor, Michigan.