AI FOR GENERAL STRATEGY GAME PLAYING

AI FOR GENERAL STRATEGY GAME PLAYING

Jon Lau Nielsen
Benjamin Fedder Jensen
Tobias Mahlmann
Julian Togelius
Georgios N. Yannakakis
IT University of Copenhagen

A JOHN WILEY & SONS, INC., PUBLICATION

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

For more technical details, please refer to the first two authors' joint masters thesis [1], or Tobias Mahlmann's PhD thesis including the detailed description of the Strategy Games Description Language [2]. Both of these are available online at http://game.itu.dk/sgdl. This research was supported in part by the Danish Research Agency project "AGameComIn" (274-09-0083).

CHAPTER 10

AI FOR GENERAL STRATEGY GAME PLAYING

10.1 Introduction

Computer strategy games[1] — games such as those in the *Civilization*, *StarCraft*, *Age of Empires* and *Total War* series, and board game adaptations such as *Risk* and *Axis and Allies* — have been popular since soon after computer games were invented, and are a popular genre among a wide range of players. Strategy games are closely related to classic board games such as *Chess* and *Go*, but though there has been no shortage of work on AI for playing classic board games, there has been remarkably little work on strategy games. This chapter addresses the understudied question of how to create AI that plays strategy game, through building and comparing AI for *general* strategy game playing.

10.1.1 Strategy games

The first documented (non-electronic) strategy games were very abstract, e.g. Chess or Go, but starting with games published in the 19th century such as Reiswitz's

[1]For brevity, we will refer to computer strategy games as *strategy games* in the following sections.

AI for General Strategy Game Playing.
By Jon Lau Nielsen, Benjamin Fedder Jensen, Tobias Mahlmann, Julian Togelius, and Georgios N. Yannakakis

Kriegsspiel did a trend take its origin which culminated in the games we play today. Originally designed for military education [3], modelling real-life conflict scenarios of various kinds, did wargames find their way into home's as a form of entertainment (e.g. H. G. Well's *Little Wars*). Within the 20th century they underwent several media transformations, from board- and tabletop-games to computer games, achieving new levels of realism and detail in the process. At this point we would like to refer the interested reader to a summary of the history of strategy games published by Sebastian Deterding [4] in 2008.

As far as we know, there is no unequivocal definition of what "strategy game" is. In the past, arguments have been made to treat (economic) simulation games, e.g. Sim City, equal to strategy games with a militaristic gameplay, e.g. StarCraft. Two examples can be seen with both Nohr and Reichert who use the theory of "Governmentality" originally by Foucault to draw similarities between simulation and strategy games [5] [6]. We however explicitly limit ourselves to militaristic games with optional supporting mechanics that model economic systems. Rather than attempting a compact definition of strategy games, we would like to enumerate a series of common characteristics of such game, bearing in mind that particular strategy games might lack one or several of these characteristics and still be part of the genre:

- The base for strategic gameplay is a topographic map that defines relations between objects and space. Positions on the map can be either given in discrete (tile-based) or real values.
- A player does not incorporate an avatar to interact with the game world. Although some games use a unit/figure to represent the player on the map, does the camera maintains a distant "bird's eye" position.
- The player interacts with the game world through the game pieces he owns. Game pieces are often distinguished between *units* (mobile-) and *buildings* (immobile objects)
- Objects on the map may have properties. Objects are divided into classes that incorporate the same attributes and abilities.
- The interaction between objects is defined implicitly through their actions and abilities.
- The computer per se only acts as a bookkeeper (or gamemaster), making the mechanics of the game invisible.
- The game requires at least two factions. Factions compete over the same or different goals. Each faction is controlled by a player. The definition of player here is transparent: it may be another local, network based, or artificial player.
- Each faction may have a different view on the game state. While there necessarily exists a well defined canon state of the game at any time, the game rules define what information is revealed to a player. This feature is often called

limited information (while games where every information is available to every player have *full information*)

To separate our definition explicitly from other "government games", such as Sim City, we further define a separation of game mechanics into *primary* and *secondary* mechanics:

- Primary game mechanics are warfare. All actions that are immediately connected to destroying, hindering, or damaging objects are considered part of this category.
- Secondary game mechanics such as economic or political processes act as support. Commonly economic mechanics are tailored towards unit production (i.e. resource gathering), and political processes evolve around diplomacy. Sometimes secondary mechanics can become crucial to the game play and decide over a win/loss.

A well-established distinction within strategy games is made between *turn-based strategy* (TBS) and *real-time strategy* (RTS) games, where the latter have a much higher granularity in both time and space, approximating continuous movement and not requiring moves to be taken every turn.

It could be argued that a well-constructed strategy game is at least as challenging and demands a similarly rich repertoire of cognitive skills to play well as does a traditional board game. Many strategy games, where the player has to decide among multiple available actions for tens of units each turn, have branching factors (number of possible actions per turn) that dwarf those of even complex board games such as Go. Some well-constructed strategy games, such as StarCraft, have seen the emergence of new and ever more complex strategies over more than a decade of highly competitive tournaments. This latter example also points to the large and growing cultural significance of strategy games. As of August 2012, there are two Korean TV channels devoted to showing competitive StarCraft matches.

Many strategy games can be played in multiplayer mode, where human players compete with each other for domination. However, for various reasons (including the time required to play a typical strategy game) many strategy games are typically played in single-player mode, against one or several computer-controlled opponents. Strategy games can also be highly useful for training and education purposes, due to their capacity for modelling real-life conflict scenarios of various kinds.

10.1.2 AI for board games

There has been extensive research done on AI for traditional board games. In particular, Chess has figured prominently in AI research from the very start, as it is easy to formalise and model, and has been thought to require some core human intellectual capacity in order to play well. Among prominent early attempts to construct chess-playing AI are Turing's *paper machine* [7] and McCarthy's *IBM 7090* [8]. Both of these used the MinMax algorithm, which builds a search tree of alternating actions

of both players up to a certain depth (*ply*) and estimates the value of the resulting board configurations at the nodes of the tree using an evaluation function. This poses the question of how to construct an accurate evaluation function. An early pioneer in using machine learning to construct evaluation functions was Samuel, whose self-learning Checkers player anticipated the concept of temporal difference learning [9].

Advances in both algorithms and computer hardware permitted a program built on the MinMax idea to win over the human Chess world champion in 1997 [10]. Subsequently, much research on board game AI shifted to considerably harder problem of playing the Asian board game Go. Go has a much higher branching factor than Chess, and it is also harder to construct a good board evaluation function, meaning that MinMax-based approaches have so far performed very poorly on Go. The current best AI approaches to Go are instead based on Monte Carlo Tree Search (MCTS), a stochastic technique that does not normally use an evaluation function [11, 12]. Like MinMax, MCTS builds a search tree each turn, but builds it incrementally, adding nodes where promising board configurations are found. Board configurations are evaluated through playing a (large) number of games from the evaluated configuration to the end of the game, choosing moves randomly. The average outcome of these playouts is used as the estimated value of the configuration.

10.1.3 AI for strategy games

Regardless of whether the purpose of a strategy game is entertainment, education or persuasion, single-player strategy games require good artificial intelligence. "Good" could in this context mean well-playing (that the player is hard for a human to win over given equal starting positions), believable (takes the sort of actions that a human player could be expected to take in the same situation), entertaining and/or player-adaptive (adapts its challenge and playing style to the player so as to optimise player experience). AI opponents that lack one or several of these properties might strike the player as artificial, predictable, too easy or too hard and lead to the player ceasing play prematurely.

Another raison d'être for good strategy game AI is procedural content generation (PCG), especially search-based PCG. Within search-based PCG, various forms of game content (such as rules, levels, items, quests and scenarios) are generated through evolutionary or other stochastic search mechanisms [13]. In strategy games, this method has previously been applied to evolve balanced maps for StarCraft [14]. An ongoing project of three of the authors of this publication is to generate complete strategy game rulesets and unit type sets [15, 16]; the Strategy Game Description Language (SGDL) used in this study is developed as part of that project. Crucially, search-based PCG requires a way of evaluating candidate content (such as maps and rules) and assigning a fitness value. Due to the complex dynamics and synergies of the various game mechanics an analytical approach seems infeasible. Furthermore it couldn't resemble various playing styles different players might show while playing strategy games. It seems therefore reasonable to play through the candidate content and base the evaluation on how the agent performed when playing the content.

For this to be possible, well-playing, computationally efficient and human-like AI is necessary.

In contrast to board games, relatively little research on AI for computer strategy games can be found in the literature. This can be seen as surprising, given the huge popularity of such games (as discussed above). A partial explanation might be the relative dearth of benchmark problems and associated software. Strategy games come and go, and the state of the art develops rapidly. Even a long-lived strategy game such as StarCraft had a shelf-life of just over ten years, whereas Chess has been around for centuries. Further, many strategy games have closed source code, lack public APIs and might be dependent on operating systems and hardware that are quickly deprecated.

One attempt to provide just such a benchmark problem is the *ORTS* (Open Real-Time Strategy) game, which includes several features common to RTS games as they appeared around ten years ago [17]. A series of competitions were held based on this benchmark, with relatively few but interestingly diverse competitors. One of them is the SORTS agent, which is an application of the famous symbolic cognitive architecture SOAR [18] to playing the ORTS game [19]. The last ORTS competition was won by Hagelbäck and Johansson's multi-agent potential field agent [20].

For some aspects of some aspect of strategy games, it is possible to use variations of classic tree search techniques such as MinMax. For example, Churchill et al. [21] use a heuristically enhanced version of AlphaBeta to play micro-scale combat scenarios in StarCraft.

However, many computer strategy games have enormous branching factors. This would seem to pose a large problem for MinMax-based approaches, and a comparative advantage for Monte Carlo approaches. MCTS has previously been applied to tactical-level gameplay in *WarGus*, an open source clone of the *WarCraft II* RTS, for multi-agent assault planning [22]. Monte Carlo techniques have also been applied to selected sub-tasks of playing the complex TBS "Civilization IV", where the stochastic action selection was additionally informed by parsing the game's manual, resulting in strategies that beat the game's built-in AI [23].

Since the high branching factor on the micro decision level is very challenging for AI systems, it may be more promising to limit their usage to macro level decisions and rely on simpler techniques, e.g. fixed finite-state machines, for actual task executions. Olesen et al. used Neuroevolution of Augmenting Topologies (NEAT), which we will discuss in more detail in section 10.4.3, to evolve agents for the game *Globulation 2*. Through dimensionality reduction based on expert domain knowledge they were able to create controllers that could adapt to a players' challenge level offline and in realtime [24].

It has also been observed that the problem of playing a complex game such as a strategy game can productively be decomposed into several layers, e.g. "strategy", "tactics" and "micromanagement" (ordering individual units around). Most of the above approaches take a monolithic view of the problem of playing strategy games, but several of them are on the other hand only tested on part of the full strategy game playing problem, for example only handling small-scale combat with perfect information and no resource collection nor building. Weber et al. constructed a StarCraft-

playing agent with a hierarchical architecture built on reactive planning, where a number of "managers" working in parallel with different aspects of the gameplay are organised in a tree [25].

As is the case AI for many other game genres, there appears to be a sharp disconnect between the approaches that have been applied to strategy games in academic research and the AI that ships with commercial strategy games. Also, just like with other types of game AI, there's preciously little (perhaps nothing) published in the academic literature about commercial strategy game AI. Informal conversations with employees at companies developing strategy games, as well as presentations at proceedingsless industry-oriented conferences, suggest that much of commercial game AI consists of rather ad hoc solutions, and often considerable cheating; in many games, the computer player would not stand a chance against a medium-skilled human player without e.g. seeing the player's hidden moves, or having troops magically appear in besieged cities.

All of the above studies consider a single game, and often just a few selected situations in a single game. There is also a shortage of studies that compare more than one approach to playing a game. Given the significant differences between different strategy games, it is possible that solutions specifically developed for a particular strategy game fail to be effective in other strategy games. Arguably, progress on AI for strategy games would be best served by an approach that compared several promising methods on several strategy games.

10.2 Research questions and methodology

In this section, we address the problem of *general* strategy game playing. This means that we want to create agents that can proficiently play a wide variety of strategy games and scenarios, not just a single game or scenario. (The definition allows the agents some time to adapt to the particular game and scenario.) We are setting ourselves this challenge so as to ensure that the contributions we make have some degree of generality, rather than just being useful hacks. At the same time, the range of games our agents are supposed to handle is considerably more constrained than the range of games expressed by the Stanford GDL, used in the General Game Playing Competition [26]; all of the games considered in this chapter feature two players moving a number of pieces each turn on a two-dimensional grid, where the pieces are capable of annihilating each other and the winning condition is to remove the opponent's pieces. All of the game rulesets (referred to as *models*) and associated scenarios used for the experiments are implemented in the Strategy Game Description Language (SGDL), a formalism and game engine capable of expressing a broad range of strategy games.

Six different agent architectures (plus variations) are implemented. These are based on techniques that have been successfully applied on various forms of game-related AI problems: playing board games (MinMax, Monte Carlo Tree Search), autonomous agent control (Neuroevolution, potential fields), commercial video game AI (finite-state machines) and strategy selection (classifier systems). Two different

random agents are also implemented for comparison. For those architectures that are based on some form of learning algorithm, relatively extensive training is performed for each agent on each model. We ensured, that each architecture was provided with an equal amount of training time.

Two different kinds of evaluation of the agents were conducted. The first was an extensive playout of every agent against every other agent on all of the defined models. From this, the relative performance (in terms of winning/losing games) of trained agents against each other can be gauged. The second evaluation form was interactive: human players played against at least two different agents each, using different game models, and answered questions about their preferences between those agents. From this we can gathered both objective data (which agent was best against human players?) and subjective (which was best-liked?) about these agents' interactions with humans.

The questions we are addressing in this chapter, and which we claim to be able to answer at least partially, are the following:

- Is it possible to construct agents that can proficiently play not just one but a range of different strategy games?
- How can we adapt some specific AI techniques that have been successful on other game-related problems to work well with strategy games?
- Which of these techniques work best in terms of raw performance?
- Which of these techniques make for the most entertaining computer-controlled opponents?

10.2.1 The SGDL and its framework

The Strategy Games Description Language (SGDL) is a model-based approach to develop strategy games. Previous approaches for other domains [26, 27] have used different paradigms, while we are formalising game mechanics as object-attribute-action relationships. This concept is commonly used in object oriented programming and its application has been discussed in the game design community [28, 29] before. The SGDL framework is an ongoing project at the IT University of Copenhagen, with the main goals of being able to express strategy games' mechanics and automatically generate them. In this section we will briefly introduce its relevant aspects for general gameplaying. More information about SGDL is available in previous publications [15, 16, 2] and on a dedicated web page[2].

The basic idea is that all *constituative* [29] rules are expressed in a tree-based model. This tree contains nodes about all game object types and their abilities and how they interact. The model also contains information about goals for every player and the creation of maps. A special game engine then interprets the model and constructs the game world and fills it with instances of objects specified in the model. Part of that process is to configure the user interface in a way the user can interact with objects in the game world. The game engine can either use its default asset set

[2]http://game.itu.dk/sgdl/

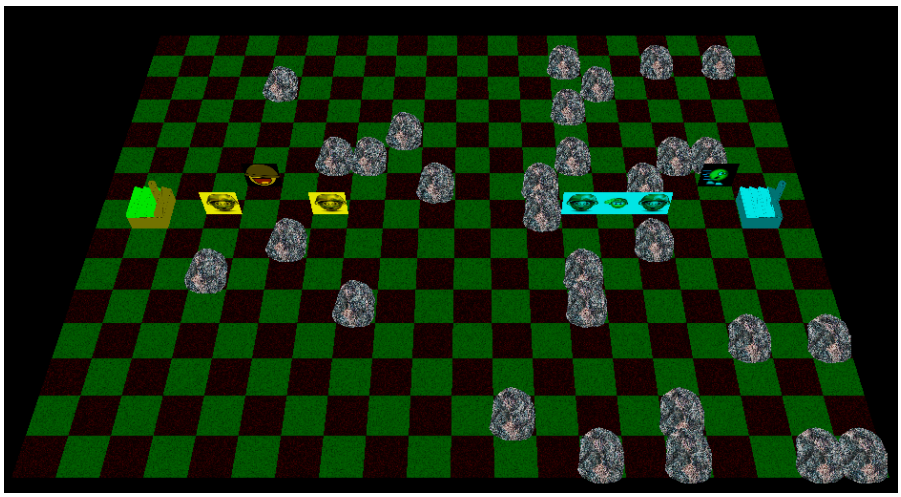

Figure 10.1 A typical screenshot from our example game "RockWars". Both players have a factory each with which to produce units, which can in turn kill the opponent's units. The rocks serve as obstacles.

(please refer to Figure 10.1 for an example) or annotations in the model that control how the game engine should visualise information (such as textures, 3D models or sounds). The game engine also provides an application interface for automated gameplay, basing on the same interaction model as the graphical client uses, i.e. artificial agents are not able to cheat by directly manipulating the game's internal data structures.

To summarise, the essential information a model must contain is:

- The map type
- Templates for all objects in the game, units and buildings alike.
- Winning condition(s)

The map is specified as a type from a given library (e.g. rectangular or hexagonal tiles), its proportions, and what properties such as resources each tile might have. However, the model does not include the topology of the map, i.e. how things are arranged on the battlefield. While a general purpose map generator for arbitrary strategy games might be possible, it is not within the current scope of our project. It would require expert knowledge about the requirements for a "good" map for a particular game. Recently Togelius et al. proposed several criteria for good maps for the game *Starcraft* [14], and Mahlmann et al. presented a map generation case study for *Dune II* [30] based on SGDL. Instead the creation of the map is controlled here by a map generator that must be written for a specific model.

Object templates are the core idea of the SGDL framework. "Object" herein refers to anything that could be placed on the map, including units and buildings owned by

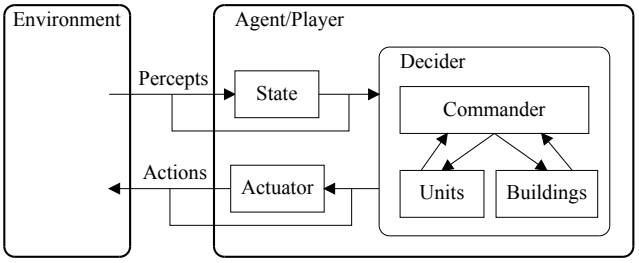

Figure 10.2 The Commander framework used for the agents in the study.

players. Each class has attributes and abilities, the latter consist of conditions that must be fulfilled before the action can be invoked, and consequences that come into effect once the action is triggered. Winning conditions are defined analogous to object actions. Both conditions and consequences are also modelled as a tree on a micro level: conditions consist of comparator nodes, and consequences contain operators/assigners.

10.3 The game, the agents and the assigner

In the games used in our experiments, each player starts with a non-moveable building which possesses the ability to spawn new units. All units take 1 turn to produce. Each unit costs a certain amount of a resource based on the SGDL model. The players' resources and the unit costs are tracked within a part of the SGDL called the *game state*. The template for this game state is read from the loaded SGDL model, i.e. the unit costs and starting resources depend on the loaded model. Each unit has one action per turn, and may select any possible action defined for its class. One of the possible actions could be to move to an unoccupied map tile or shoot at an enemy unit. If a unit loses all its health points it is removed from the game. If a player loses all his units (besides his factory) and has no resources left to produce new units, his opponent wins the game. Should the game take longer than 100 turns, the game is a draw.

The agents in the study are based on a hierarchical agent framework named the "Commander framework" based on the *Intelligent Agent Architecture* by Russel and Norvig [31], as can be seen in Figure 10.2. The framework consists of a commander entity on the highest logical layer, and sets of unit and building entities. Objects in a game belonging to an agent are linked to these sets of unit and building entities in the framework, while the commander entity is a strategic entity only. Thus the framework is separated in a higher level layer, called the strategic layer, and a lower level layer, called the unit layer.

The framework was designed such that it can be used by game tree based techniques as well as multi-agent techniques. This is possible by the two-way commu-

nication between all types of entities, which allows a commander to control units and/or buildings, units and buildings to act autonomously or any other combination necessary.

The communication with the SGDL game engine is maintained through two utility systems named State and Actuator. Because the SGDL model of a game consists of an unknown quantity of percepts and actions with unknown properties, it is beneficial for non game tree based techniques to use a system that categorises percepts and actions into sets of known quantities and properties. An agent may test if a specific action is possible at any time during his turn by testing its conditions. The SGDL framework also supports supplying all possible actions for a certain unit. This is done through testing all the conditions of all of the actions that object could theoretically invoke. Actions which require an additional object (target object) are tested with all objects that are also on the map. Because of a pre-set constraint, no action in our games requires more than two objects (acting object and target object) we can limit the search to one extra object. Otherwise conditions would have to be checked against all permutations of objects on the battlefield. Although an agent with more domain knowledge might apply a faster and more efficient way to select actions, the agents described in this chapter rely on the set of possible actions created through the described "brute-fore" search method. If an agent submits an action to the framework that is not possible, it would simply get denied.

The State system consists of a set of general information that captures a subset of the percepts thought to be the minimum amount of information necessary for agents to react meaningfully to the game environment. Included in the set of states are the type of class of the unit, its health, the distance, angle and relative power of the nearest three opponents, the distance and angle of the nearest two obstacles and the distance and angle of the opponents building. The Actuator system uses a one ply game tree search in order to determine the effect of all given actions, and categorises them into a finite set of actions with known effects. Included in the set of actions are the attack actions that do the most damage to opponents, actions that kill opponents and actions that cause movement in one of eight possible directions. The disadvantage of these systems is that information is no longer complete given the categorisations made, but they decrease the search space by a very large magnitude; a requirement for many techniques to do meaningful searches.

Another utility function was developed for commander entities, which can provide additional information in terms of a relative measurement of power of unit objects relative to each other and can assign orders to unit entities on the lower layer. Relative power information is gained through short simulations of the unit object types against each other, where power is based on the steps it takes for one to kill another. The order assignment is done through a neuroevolutionary approach based on NEAT [32]. A bipartite graph that consists of the set of units belonging to an agent are fully connected to a set of units that belong to the enemy. Each edge of the bipartite graph is weighted by a neural network evolved through NEAT with a set of information relevant to each edge, i.e. distance, health and relative power measurement between the units connected. Assignments are determined by hill climbing,

Table 10.1 Unit properties for SGDL models

	SGDLs				
Unit properties	Chess	Shooter	Melee	RPS	Random
Random cost	✓	×	✓	✓	✓
Random health	✓	×	✓	✓	✓
Random ammo	✓	×	✓	✓	✓
Random damage	✓	×	✓	✓(special)	✓
Random range	×	×	×	✓	✓
Movement note	Special	1-step	1-step	1-step	1-step

where the highest valued edges of the bipartite graph are selected for each unit that requires an assignment.

To test the agents' flexibility with different SGDL models, five distinct models were created to represent different strategy gameplay aspects. As seen in table 10.1, the models were named *chess*, *shooter*, *melee*, *rock-paper-scissor (RPS)* and *random*.

- *Rock-paper-scissors (RPS)*: a balanced strategy game where each unit can do heavy damage to one other class of unit, light damage to another and no damage to the third. This mirrors a popular configuration in strategy games where tanks are effective against foot soldiers, foot soldiers against helicopters and helicopters against tanks. All units have a movement of one.
- *Melee*: Similar to the RPS model, but all units have an attack range of one, forcing them to chase and entrap each other.
- *Shooter*: Perhaps to the model that is most similar to a standard strategy game. Shooter has 3 different classes, a sniper, a soldier and a special operations agent (special ops). The sniper has high range, medium damage and low health, the soldier has medium range, low damage and high health and the special ops has low range, high damage and medium health.
- *Random*: Units are only able to move one step, and the cost, health, ammo and damage of each class against all others is randomised for every game.
- *Chess*: A simplified chess game with unit movements and capabilities inspired by the rook, knight and bishop pieces.

The properties of all the models used are summarised in table 10.1. There was a limit of 100 turns per game and a limited amount of units which could be built based on their cost. The players started with an equal random amount of money that could be spent on units. The games are turn based, and there is no fog of war. The models described above determine the rules of each game. Any action, be it movement, shooting or similar, constitutes a turn, and so does doing nothing. Units are symmetric for the players in all games regardless of models and maps.

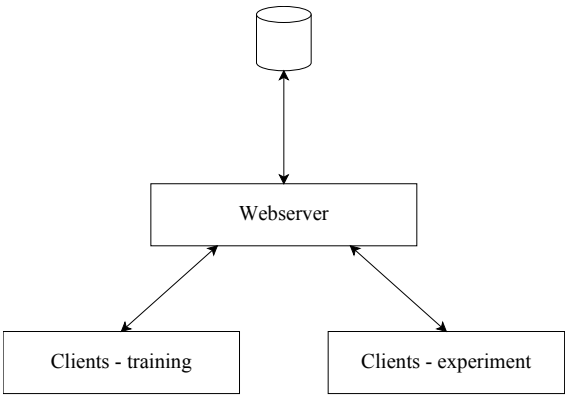

Figure 10.3 Client-server system used for training and experimentation.

10.3.1 Implementation details

The software was developed as a console Java application, relying on a number of commonly used open-source Java libraries, in particular *Apache Commons*, *Log4J*, *SqLite* and *XStream*. It exposes the framework and the AI agents to a web server as illustrated in Figure 10.3. The client-server system allows clients to train and conduct experiments, and the web server to collect the data generated by the clients. Not only did it allow for a real-time view of the training or experimentation progress, but it also ensured that data was safely collected and classified. Client-server communication was done through the HTTP protocol by GET and POST requests to various PHP scripts that fetched or updated data in a MySQL database server-side. The models, maps and training data was stored server-side and requested by the clients, and the clients responded with results.

It was deployed to approximately 50 computers installed in lab rooms at IT University of Copenhagen. All computers ran Windows XP and were based on Intel Core 2 Duo processors. Most experiments were run over night, watched over by the two first authors. We did not keep track of the exact time consumption for the experiments, but had we done so the processing time would be measured in weeks or tens of weeks.

10.4 Agents

We created eleven agents based on several different techniques as presented in Table 10.2. The non-learning agents' main purpose was to serve as training partners for the evolving agents, but were also included in our experiments. The following sections will cover the details of each agent implemented.

Table 10.2 Agents in the study

Agent name	Purpose
Random	Opponent in agent test
SemiRandom	Training, Opponent in agent test
FSM	Training, Opponent in agent test, Human play testing
NEAT	Agent versus Agent testing, Human play testing
NEATA	Agent versus Agent testing
MinMax	Agent versus Agent testing, Human play testing
MCTS	Agent versus Agent testing, Human play testing
PF	Agent versus Agent testing, Human play testing
PFN	Training
XCS	Agent versus Agent testing
XCSA	Agent versus Agent testing

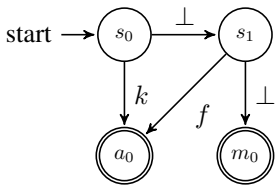

Figure 10.4 Finite-state automata of the SemiRandom agent units

10.4.1 Random action selection

Two agents that rely on random action selection were created in this study to train the evolving agents and to provide a performance baseline. Both agents are capable of fitting into a multi-agent framework, as the logic executes on the level of the individual unit and not at a strategic level. These two agents are the *Random* agent and the *SemiRandom* agent.

The Random agent selects a random action from the set of possible actions given by the Actuator, resulting in random behaviour. The SemiRandom agent is designed to move randomly but use the best available offensive action possible, thus making it an offensive but largely immobile opponent. The agent uses the Actuator as well, which guarantees that the most effective actions in the game are used. As can be seen in Figure 10.4, a *finite-state automaton* or *finite-state machine* [33] is used to determine the action to perform.

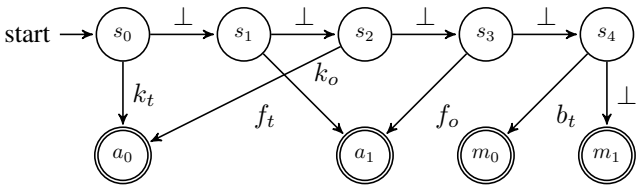

Figure 10.5 Finite-state automaton of the FSM agent's units

10.4.2 Finite-state machine

Similar to the random agents, the *finite-state machine* (FSM) agent was created to provide a performance baseline and a training partner for other agents. It utilises a finite state machine architecture with movement selection based on a local breadth first search. Figure 10.5 shows the structure of the automaton for the units, where the Actuator is used for action selection analogue to the random agents. The FSM agent is an effective opponent, but it requires hand-coded domain knowledge about the game model.

Congruent to the SemiRandom agent, an ordered sequence $\langle s_0, \ldots, s_4 \rangle$ of transitional states is traversed. Unlike the SemiRandom agent, the FSM agent requires a hostile target for each individual sub-agent in order for a search to be successful; it is also used for attack selection as the target is prioritised. The accepting states are $\{a_0, a_1, m_0, m_1\}$, and are given by the Actuator. Since the FSM agent favours a targeted unit, the kill and attack conditions k and f are subdivided into k_t, k_o and f_t, f_o where t is the target, and o is any other unit. The breadth-first search is designated as b_t, where the b_t condition is true if the breadth-first search is able to find a path towards the target. In this case the accepting state m_0 selects the first movement action which leads along this path.

The breadth first search is local because it has a limited amount of movement actions that can be searched; an *expansion limit*. In order for the search to be effective, a distance heuristic was applied on the ordered sequence used for expansion of moves in the search. When the limit has been reached the search terminates and the action with the shortest distance to the target is executed.

10.4.3 Neuroevolution of augmenting topologies

The *Neuroevolution of Augmenting Topologies* (NEAT) agents are based on the evolutionary algorithm for neural networks developed by Stanley and Miikkulainen [32]. This algorithm has previously been used with success for evolving agent controlling neural networks in, but not limited to, shooter games and racing games. The technique is a form of *topology and weight evolving artificial neural network* (TWEANN), such that it not only optimises weights in a neural network, but also constructs the network structure automatically via artificial evolution. Within the NEAT agents, action selection is based on an artificial neural network that has been trained through

machine learning using evolution. A fitness function evaluates the performance of genomes in a population, and the fittest members are subsequently selected for the creation of new members by combining their genetic information through a crossover genetic operator [34, 35].

Given the nature of artificial neural networks that they can approximate any function given an arbitrary large network [36], and a topology which evolves to a functional structure, the agents are able to learn general gameplaying depending only on the fitness function. However, in this implementation the State and Actuator utilities were used to simplify and normalise the number of inputs and outputs. This means that the technique operates on a subset of the actual state and action space of the games, as was discussed in section 10.3.

The two following agents have been created that use an artificial neural network evolved through NEAT:

1. NEAT agent (Neuroevolution of Augmenting Topologies agent)
2. NEATA agent (Neuroevolution of Augmenting Topologies agent with Assigner)

The fitness function used for the successful NEAT agent - out of several investigated in the study - can be seen in equation (10.1). Here w is the amount of wins, l is the amount of losses, and d is the amount of draw games. Each genome is tested in six games against three opponents, and evaluated using this function. The function was made to force the agent into avoiding draw games and prioritise winning. However, its behaviour is consistently hesitant to pursuing opponents, and instead waits for the opponent to approach.

$$f_{NEAT}(a_i) = \frac{w}{w + l + d} \tag{10.1}$$

Several fitness functions were investigated, using more information than just the winrate as above, such as including a normalised distance measure to encourage a behaviour which engages opponents more; a flaw of the above fitness measure. However, the winrate decreased when using these information additions, even though the behaviour of the agent became more as desired in that it would aggressively pursue the opponent. The problem might be caused by conflicting objectives; pursuing seems to counteract its ability to win. Equation (10.1) received the largest amount of victories, and was thus chosen for training.

The NEATA agent has one variant which can be seen in equation (10.2). Here s is the number of successful orders carried out by units given out by the Assigner (see section 10.3), b is the number of kills that were not given by orders and q the number of failures to perform orders, e.g. the death of a friendly unit. It is normalised to the number of units which have received orders.

$$f_{NEATA}(a_i) = \left| \frac{s + \frac{b}{4} - \frac{q}{4}}{u} \right| \tag{10.2}$$

The function drives the agent to evolve a behaviour that can successfully kill the hostile unit which has been designated as a target, and potentially kill any other

hostile unit it encounters on its way. Because of the negative value given for a failure, it also attempts to avoid destruction while carrying out the orders.

10.4.4 MinMax

The MinMax agent is based on the classic MinMax algorithm with alpha-beta pruning [31], which is one of the simplest and most popular algorithms for playing games and which has achieved considerable success on board games with low branching factors, like Chess and Checkers.

When implementing game tree search-based agents, it was decided that every branch in the search tree represents a set of actions, one action for each friendly moveable unit. We will refer to such a set as a *MultiAction* in the following.

Both game tree agents run in the Commander-Structure within the Commander framework as seen in Figure 10.2. After a search has been executed, the best MultiAction is determined and its actions are distributed to the units. Neither the MinMax nor the Monte Carlo Tree Search (MCTS) agent (presented in the following subsection) use State or Actuator as the algorithms search ahead through potential actions.

Implementing MinMax with alpha-beta pruning into an agent required the modification of the algorithm as its runtime complexity grows rapidly with the branching factor and depth of the tree. Since the amount of units and actions are unknown in the model, a limit had to be placed on the amount of MultiActions possible. Without a limit, too many MultiActions could cause the agent to play poorly or become dysfunctional.

To limit the MultiActions, depth first search (DFS) is performed on a tree with moveable units and actions. Please refer to Figure 10.6 as an example, where U_x is moveable unit x, and U_{xy} is its action y. The DFS is limited to only choose from the child nodes of its current best choice, starting from the root. For example, should the DFS choose $U_{1attack}$ over U_{1flee}, it would then have to choose between $U_{2attack}$ and U_{2flee}. When a leaf is found, the MultiAction (built from path of actions selected from root to leaf) is saved, and the DFS moves one step back towards the root and selects again.

To help guide the selection of MultiActions, a heuristic is needed to evaluate how advantageous a game state for an agent is. The heuristic used is the same board evaluator used within MinMax when a maximum depth is reached. Constructing a heuristic for changing models proved to be a challenge as units in class 1 might have different actions and attributes in different SGDL models. A static heuristic would be near impossible to construct and instead a neural network was evolved using NEAT with the inputs: the health ratio between friendly and hostile units and the average euclidean distance to nearest enemy, weakest enemy, friend and enemy building. The single output of the network is how favourable the board configuration is. The neural network was evolved by evaluating its performance based on its win rate against a set of test agents of varying behaviour.

To help MinMax predict the actions of the enemy, even without units currently on the board, the buildings and their actions had to be implemented in the tree seen in Figure 10.6. This would further increase the amount of MultiActions if every

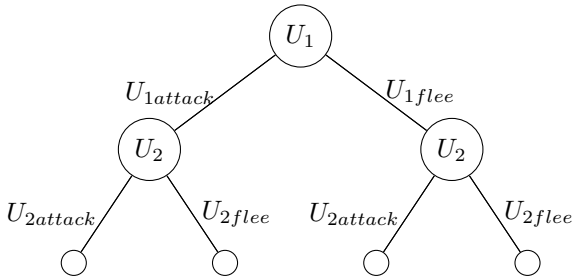

Figure 10.6 Action tree used to find MultiActions, performed at each node in the search tree.

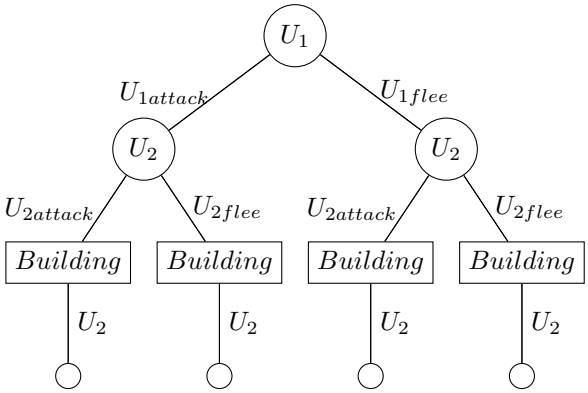

Figure 10.7 Action tree illustrating the use of buildings. The building nodes are allowed only one child each to limit the complexity.

action of the building is represented. It was therefore chosen to limit the buildings' actions to one. The unit produced in the search tree would be decided randomly (see Figure 10.7).

10.4.5 Monte Carlo Tree Search

The Monte Carlo Tree Search (MCTS) agent is based on the MCTS algorithm, which has recently seen considerable success in playing Go [11]. Even though MCTS is known for handling trees with large branching factors, the branching factor of most SGDL models is drastically higher than Go. Considering this issue, the MCTS agent was implemented with the same MultiAction filter as the MinMax agent. Once MCTS was implemented, it could be observed that the algorithm only produced 100-150 Monte-Carlo simulations per second due to the computational overhead of cloning in Java. As a solution to this, MinMax's board evaluation function was used

instead of the *play-out* phase. The regular play-out outcome z_i of simulation i is replaced with a state value approximation, which is backpropagated towards the root as normal. The Monte-Carlo value (Q) can be seen in equation (10.3), where Γ is an indicator function returning 1 if the action a was selected in position s at any of the i steps, otherwise 0, $N(s,a)$ is the amount of simulations through s where action a was chosen, and $N(s) = \sum_{i=1}^{|\mathcal{A}(s)|} N(s_i, a_i)$, where $\mathcal{A}(s)$ is a finite set of legal actions from state s. The change of playout-phase in MCTS gave a significant improvement in Monte-Carlo simulation count.

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{N(s)} \Gamma_i(s,a) z_i \tag{10.3}$$

Several tree policies were tested such as UCT [37] in Equation (10.4), progressive bias [38] in Equation (10.5), Monte-Carlo Rapid Action-Value Estimation (MC-RAVE) in Equation (10.6), (10.7) and (10.8) and UCT-RAVE in Equation (10.9) [39, 40].

UCT solves the exploration dilemma by utilizing the UCB1 [41] algorithm by scaling the exploration factor c, so the amount of exploration can be limited.

$$Q_{UCT}(s,a) = Q(s,a) + c\sqrt{\frac{\log(N(s))}{N(s,a)}} \tag{10.4}$$

Progressive bias is an added heuristic to the standard UCT heuristic to guide the search. The impact of the heuristic lessens as simulations through state s using action a increase.

$$Q_{pbias}(s,a) = Q(s,a) + c\sqrt{\frac{\log N(s)}{N(s,a)}} + \frac{H_{(s,a)}}{N(s,a)+1} \tag{10.5}$$

The RAVE values in MC-RAVE quickly converges a bias value $\tilde{Q}(s,a)$ for action a from the subtree of the node representing state s. Since this value is biased, MC-RAVE uses a decreasing factor $\beta(s,a)$ relying on a k-value to determine how fast the factor decreases. Sylvain Gelly and David Silver found the highest win rate in Go using a k-value of 3000 [39]. Due to lower MCTS iterations, the k-value had to be lowered in the experimentations to faster rely on the actual Monte-Carlo values and not the biased RAVE-values. Because the MCTS agent used a heuristic, the biased RAVE-values were evaluations from subtrees instead of actual playout values.

$$Q_{MCRAVE}(s,a) = \Big(\beta(s,a)\tilde{Q}(s,a) + (1-\beta(s,a))Q(s,a)\Big) \tag{10.6}$$

$$\beta(s,a) = \sqrt{\frac{k}{3N(s)+k}} \tag{10.7}$$

$$\tilde{Q}(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{\tilde{N}(s)} \tilde{\Gamma}_i(s,a) z_i \tag{10.8}$$

UCT-RAVE adds the exploration factor $c\sqrt{\frac{\log N(s)}{N(s,a)}}$ from UCT to MC-RAVE.

$$Q_{UCTRAVE}(s,a) = Q_{MCRAVE}(s,a) + c\sqrt{\frac{\log N(s)}{N(s,a)}} \tag{10.9}$$

When testing the algorithms, MC-RAVE showed the highest win rate best with k-value of 10. During the experiments, a pattern seemed to emerge. UCT-RAVE ($c = 0.1$) scored a worse win rate than MC-RAVE against FSM (38.95% vs. 45.96%), SemiRandom (32.63% vs. 37.89%) and Random (56.49% vs. 66.14%) with p-values 0.05, 0.16, and 0.02. For the MCTS agent, UCT ($c = 0.1$) performed worse than UCT ($c = 0$). It seemed when MCTS iterations were forced to explore, rather than focusing on the early best looking child nodes, the win rate was decreasing. This is most likely caused by either a too low iteration count and/or the use of a board evaluation function, replacing the regular play-out phase. If the reason is a too low iteration count, giving the algorithm more time to think (more than one second) would increase the iterations and might as a result possible reward the act of exploring child nodes of less immediate interest. On the other hand, raising the time constraint to more than a second seems not desirable, as it most likely would affect the experience of a human player in a negative way; even though we are only considering turn-based games. Also, due to replacing the playout-phase with a neural network evolved using NEAT, might affect the Monte Carlo-value by setting it close to its exact value even after only a few iterations - and exploration would therefore become obsolete.

10.4.6 Potential fields

The potential field (PF) agent developed in this chapter is similar to the multi-agent potential field approach which has recently shown good performance in some real-time strategy games [42, 43, 44]. The potential of a point on the map of the game is expressed as in equation 10.10, where P is a set of potentials and $w(p_n)$ is a function that maps a weight to potentials and the pheromone trail. Potential functions take a distance from the x and y variables and the position of the potential p_i using the euclidean distance. A pheromone trail is given as k, which is a trail of pheromone left by each unit, where each individual pheromone is weighted inside the pheromone function, such that they decrease in strength over time. As such, they serve as a negative trail of potentials with decreasing effect, and forces the units to move in paths not previously taken. There is a potential for each object on the map which contains the position of the object, and additional input variables not given in equation 10.10 that apply specifically for the various potentials depending on their purpose.

$$f(x,y) = \sum_{i=1}^{|P|} \left(p_i\left(d\right) w(p_i)\right) + \left(k(x,y)w(k)\right) \tag{10.10}$$

By using this formula to calculate the potential of a point, it is not necessary to calculate the global field of the map. Each unit contains its own potential field, which is calculated for the legal moves that it can make in its turn, and in order to keep the pheromone trail local to the unit.

$$p_{hostileunit}(d) = \begin{cases} \left(\frac{|m-d|}{m}\right)^2 power, & \text{if } power > 0; \\ -\left(\frac{|m-d|}{m}\right)^2 \frac{1}{2}, & \text{otherwise.} \end{cases} \tag{10.11}$$

As there is a potential for each object, and given that there are different types of objects, multiple potential functions such as the one in Equation (10.11) were formulated. The above function creates a potential for hostile units, where m is the maximum distance on the map, d is the distance between the sub-agent and the hostile unit, and $power$ is the relative power measure given by the Assigner utility. Various other functions are given for friendly units, buildings obstacles etc.

An additional agent named PFN with a negative sign in Equation (10.11) was used for training, as it would avoid enemy contact and require the agents trained to learn how to give chase.

10.4.7 Classifier systems

Two agents were implemented based on eXtended Classifier Systems (XCS) [45]: a regular *XCS agent* and a XCSA (eXtended Classifier System using Assigner) agent using the Assigner for orders. Both agents operated within the units and are not using the Commander entity in the Commander Architecture as seen in Figure 10.2. All units for both agents shared the same XCS structure, resulting in shared information about the environment.

The XCS classifier system builds on Holland's Learning Classifier Systems [35] (LCS) which is a machine learning technique that combines reinforcement learning and evolutionary computing. A classifier system creates rules through evolutionary computing and tries to predict the external reward by applying reinforcement learning through trial and error. LCS changes the fitness of the rules based on external reward received, while XCS uses the accuracy of a rule's prediction.

To adapt to changing SGDL models, the XCS structure was slightly modified. In Wilson's XCS a covering occurs when the amount of classifiers in the *Match set* is below a threshold. Following Wilson's advice by populating through covering, setting such a threshold can be difficult with changing SGDL models, as the amount of actions are unknown. A low threshold resulted in the Match Set filling up with move actions, as attack actions were met later in the game when opponents were engaged. The threshold was changed to force the XCS structure to have at least one classifier for each possible action in the current environment.

In some SGDL models, unit attributes changed over different games, therefore classifiers representing illegal moves are removed from the Match Set.

To reward the *Action Sets*, the XCS agent had to wait for the game to end, in order to receive a *won*, *draw* or *loss* from the game. All Action Sets a were then rewarded through Equation 10.12. There, Ω is a function returning 1 for win and 0 for loss or draw, D is the average euclidean distance to nearest enemy and D_{max} is the maximum possible distance.

$$r(a) = 1000\Omega + 500(1 - \frac{D}{D_{max}}) \tag{10.12}$$

The XCSA agent utilised the Assigner's order updates throughout the game and rewarded (see Equation 10.13, where Λ is the is the euclidean distance to the target) immediately once it was told if the action was good or bad. Using Equation 10.13, each order event signal was rewarded differently. Upon receiving a successful event signal, the $reward_{order}$ was set to 1000. A mission cancelled or failed signal led to $reward_{order}$ being 0, and should the unit receive the event signal of killing an enemy outside the ordered target, 500 were set for $reward_{order}$.

$$r(a) = reward_{order} + 500(1 - \frac{\Lambda}{D_{max}}) \tag{10.13}$$

10.5 Results of agent versus agent testing

Before evaluating them, most of the agents needed to be trained in order to perform well. Training was undertaken separately for each agent on each model, but always against all three artificial opponents. It was ensured that all agents were trained for the same amount of time (for fairness), and long enough, so a performance convergence could be observed for each agent type.

The performance of the agents against the FSM, SemiRandom (SR) and Random (R) agent in terms of the ability to win games on the different SGDL models and maps was measured through experimentation. Nearly a million games in total were run, concluding in the results presented below.

The results are analysed in terms of the *win rate* (WR) and the *win loss ratio* (WLR), where the win rate is given as $\frac{w}{w+l+d}$ and the win loss ratio is given as $\frac{w}{w+l}$. Here w is the amount of games won, l is the amount of games lost and d is the amount of games that ended in a draw. The win loss ratio ignores the draws, in order to focus on the ratio of win/loss against opponents, but must be seen with respect to the win rate which is the win ratio in terms of total games played.

In Table 10.3 the following terms are used; *Opp.* refers to the opponent agents, *W* refers to won games, *L* refers to lost games, *D* refers to draw games, *WLR* refers to the win loss ratio, *WR* refers to the win rate. The standard deviations are given in Table 10.4 where the following terms are used; $\overline{\sigma_{WLR}}$ refers to the mean standard deviation of the win loss ratio and $\overline{\sigma_{WR}}$ refers to the mean standard deviation of the win rate. The terms $\sigma_{\overline{\sigma_{\mathrm{WLR}}}}$ and $\sigma_{\overline{\sigma_{\mathrm{WR}}}}$ denote the standard deviation of the population

Table 10.3 Summary of agent versus agent results

Agent	Opp.	W	L	D	WLR	WR
MinMax	FSM	3667	3264	1619	52.91%	42.89%
MinMax	SR	2164	1584	4802	57.74%	25.31%
MinMax	R	3787	297	4466	92.73%	44.29%
MCTS	FSM	4038	2799	1713	59.06%	47.23%
MCTS	SR	2549	947	5054	72.91%	29.81%
MCTS	R	3930	225	4395	94.58%	45.96%
XCS	FSM	16691	31865	7019	34.37%	30.03%
XCS	SR	2695	5337	47543	33.55%	4.85%
XCS	R	6226	1570	47779	79.86%	11.20%
XCSA	FSM	13395	35280	6900	27.52%	24.10%
XCSA	SR	2653	5771	47151	31.49%	4.77%
XCSA	R	6622	1679	47274	79.77%	11.92%
PF	FSM	25505	23643	6427	51.89%	45.89%
PF	SR	11526	14461	29588	44.35%	20.74%
PF	R	29711	1976	23888	93.76%	53.46%
NEAT	FSM	26461	21741	7373	54.90%	47.61%
NEAT	SR	4172	4496	46907	48.13%	7.51%
NEAT	R	9759	1393	44423	87.51%	17.56%
NEATA	FSM	20391	28308	6876	41.87%	36.69%
NEATA	SR	2973	8122	44480	26.80%	5.35%
NEATA	R	6726	2901	45948	69.87%	12.10%

Table 10.4 Standard deviantions of agent versus agent results

Agent	Opp.	$\overline{\sigma_{\text{WLR}}}$	$\overline{\sigma_{\text{WR}}}$	$\sigma_{\overline{\sigma_{\text{WLR}}}}$	$\sigma_{\overline{\sigma_{\text{WR}}}}$
MinMax	FSM	0.05	0.04	0.02	0.02
MinMax	SR	0.08	0.04	0.04	0.02
MinMax	R	0.04	0.04	0.03	0.02
MCTS	FSM	0.04	0.04	0.02	0.02
MCTS	SR	0.07	0.04	0.02	0.02
MCTS	R	0.04	0.04	0.04	0.02
XCS	FSM	0.02	0.02	0.01	0.01
XCS	SR	0.05	0.01	0.01	0.00
XCS	R	0.04	0.01	0.03	0.01
XCSA	FSM	0.02	0.01	0.01	0.01
XCSA	SR	0.04	0.01	0.02	0.01
XCSA	R	0.05	0.01	0.03	0.01
PF	FSM	0.02	0.02	0.01	0.01
PF	SR	0.03	0.01	0.01	0.01
PF	R	0.01	0.02	0.01	0.01
NEAT	FSM	0.02	0.02	0.01	0.01
NEAT	SR	0.04	0.01	0.02	0.00
NEAT	R	0.03	0.01	0.02	0.01
NEATA	FSM	0.02	0.01	0.01	0.00
NEATA	SR	0.04	0.01	0.02	0.00
NEATA	R	0.04	0.01	0.03	0.00

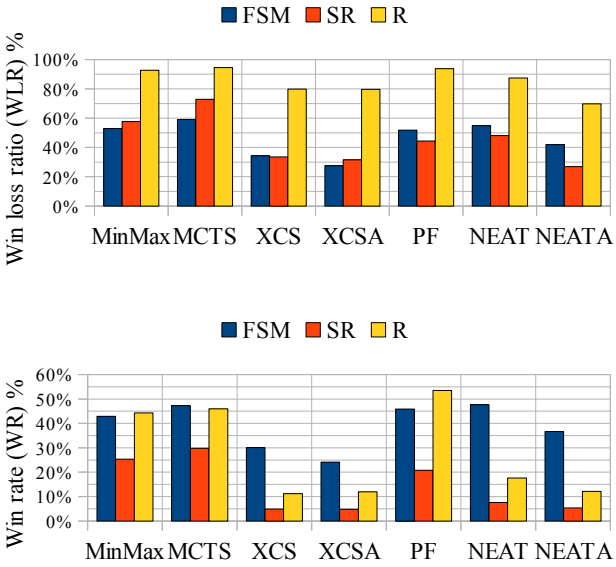

Figure 10.8 Summary of agent versus agent results

of the standard deviations given in the above means. This is necessary as the samples are divided on different models and maps.

In total, 8550 games were played for the adversarial search based agents MCTS and MinMax and 55575 for the other agents. The previously mentioned computational complexity of the search based agents required us to make this differentiation in order to perform the experiments in a reasonable time frame. The variation between samples, as seen in Table 10.3, is highest with MinMax and MCTS given the smaller sample size, but it is low in general for all agents.

As can be seen in Figure 10.8 and Table 10.3, the MCTS, MinMax, PF and NEAT agents have a WLR near or above 50%. XCS, XCSA and NEATA have a WLR lower than 50% on all opponents other than the Random agent. Only the MCTS and MinMax agent were able to defeat the SemiRandom agent. This may be because the SemiRandom agent demonstrated quite good gameplay on most models. It tends to gather its units in small clusters with effective selection of offensive actions based on the finite-state automaton logic.

With regards to WR, most agents had a performance less than 50% against all opponents because of draw games. The MinMax, MCTS and PF agents have the highest performance in general in terms of their WLR as noted above, and a low number of draw games compared to the other agents. The NEAT agent has a very low WR, which is caused by a very high amount of draws. This is due to its behaviour, which is similar to the SR agent, that it gathers in clusters near its spawn, and waits

for the opponent. Breaking the turn limit of 100 results in a high amount of draws against the SR and R agents which, in general, approach their opponent rarely. It does however defeat the FSM agent, as it is built (via its incorporated BFS method) to engage in a battle.

The XCS, XCSA and NEATA agents have a performance which was below the chosen acceptable threshold of a WLR of 50% against the three opponents, and an equally poor WR performance in terms of a high amount of draws games as well.

In conclusion, the MinMax, MCTS, PF and NEAT agents were determined to be adequate in various models and map combinations, thus capable of general gameplay.

10.6 Results of human play testing

To test how enjoyable, human-like and challenging the agents were, we set up an online user test system. Through the system human participants were paired up for a *Random* game, after a short tutorial, with an agent and then for a second round with another agent. Only the FSM, MinMax, MCTS, PF and NEAT agents were used in this test. After the games a short survey was presented where players could report their preferences regarding the opponent, game and the experiment itself. The following four questions were asked after the two games were played:

1. Which opponent was more challenging?
2. Which opponent was more enjoyable to play against?
3. Which opponent played more like a human?
4. Disregarding the opponent, which game did you prefer?

All questions could be answered with either A, B, Neither or Both, where A and B refer to the first and the second game session. The numbers presented in this section are based on these self-reports.

The total number of participants was 60. The average age was 23.47 years and 95% of the participants were male. All participants played computer games in general and all participants enjoy playing strategy games. Of the participants, 45% play games for fifteen or more hours a week, 26.67% play games between ten to fifteen hours a week, 18.33% play games between ten to six hours a week and 8.33% play games between five to one hour a week. 23.33% consider themselves experts in strategy games, 55% consider themselves advanced players and 20% consider themselves novices. One participant did not answer how many hours she plays games, or what her self-assessed skill level was. It may be noted that the selection of participants is heavily biased towards young male experienced gamers, but given that most of the players were recruited in an online community for strategy games, we considered this demographic as the core audience for our experiment and this was not incorporated into our analysis; the effect of self-selection can therefore be neglected.

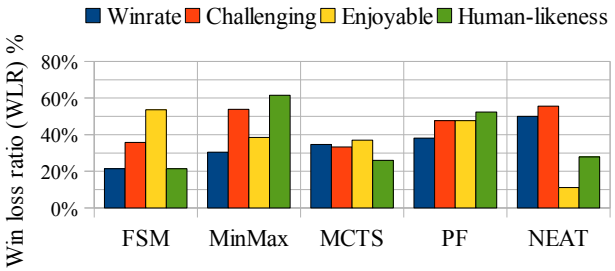

Figure 10.9 Summary of human play results

Table 10.5 Summary of human play results

Agent	Games	Win rate	Challenging	Enjoyable	Human likeness
FSM	28	21.43%	35.71%	53.57%	21.43%
MinMax	26	30.43%	53.85%	38.46%	61.54%
MCTS	27	34.62%	33.33%	37.04%	25.93%
PF	21	38.10%	47.62%	47.62%	52.38%
NEAT	18	50.00%	55.56%	11.11%	27.78%

As can be seen in Figure 10.9 and Table 10.5, the agent with the highest win rate against humans was the NEAT agent with a win rate of 50%. The worst in terms of win rate was the FSM agent with a win rate of 21.43%, which was in line with the results gathered in the experiments against non-human opponents.

In terms of challenge, the NEAT agent was the most difficult with a challenge rating of 55.56%, and the least challenging was the MCTS agent with 33.33% followed by the FSM agent with 35.71%. The MinMax and PF agents are above or near 50% in terms of participants who fought the agents and found them challenging.

The most enjoyable agent was the FSM agent: 53% of the participants who fought the agent found it enjoyable. The least enjoyable opponent was the NEAT agent with only 11.11% of the participants rating it as such. The PF was found the second most enjoyable agent with 47.62%, and both MinMax and MCTS were found nearly equally enjoyable.

In terms of human-likeness, the most human-like reported agent was the MinMax agent with 61.54% and the PF agent with 52.38%. The MCTS and NEAT agents were not perceived human-like with only 25.93% and 27.78% respectively. The least human-like agent was the FSM agent.

Although the NEAT agent was the best performing in terms of win rate and challenge provided for the participants, players reported it as less enjoyable or human-like. This can be explained by its passive behaviour, not approaching the enemy; some players even reported it as "broken". This could also be observed in the tests against non-humans, where it accumulated a large number of draws against the SemiRandom and Random agents. The second best performing agent, in terms of win rate and challenge, was the PF agent. It also provided a good level of enjoyment for the participants, and was considered fairly human-like. The third best performing agent was MinMax in terms of win rate and challenge, and provided a good level of enjoyment for the participants, as well as being perceived as the most human-like. The MCTS agent provided a better performance in terms of win rate and challenge than the FSM agent, and was perceived more human-like. However, the FSM agent was despite its low performance the most enjoyable agent of all five in the experiment.

Although the data is not completely conclusive, it shows that the agents based on MinMax, MCTS, Potential Fields and NEAT performed better than the simple FSM agent in terms of win rate, challenge and human-likeness. Analogue, those agents showed a superior performance in terms of win rate and win loss rate against the Random and SemiRandom agents in the non-human experiments. The only exception is the NEAT agent, which was unable to approach SemiRandom and Random due to the reasons discussed. It can therefore be concluded from the non-human and human experiments that the agents based on MinMax, MCTS and Potential Fields have high skills in terms of their ability to play, that they are flexible under changing rule-sets and capable to some degree at showing human-like behaviour. Given that all agents perform a turn in less than a second for all models, we can state that all agents have shown a reasonable runtime behaviour for our needs.

The NEAT agent was not enjoyable for the participants and it was not perceived human-like. It was also unable to engage the Random and SemiRandom agents. Therefore it cannot be considered well playing in general, but has shown potential

in defeating skilled opponents. The Assigner decreased the performance in agents when used to assign orders, but its ability to measure the relative power of enemy units was beneficial. The XCS, XCSA and NEATA agents have shown a poor level of skill in play against any opponent than the Random agent, both in terms of win rate and win loss rate.

10.7 Discussion

While we have taken all the efforts that were reasonable within the time frame of the project to ensure that all the agents were compared fairly, a complete fair comparison is in our opinion hardly even a theoretical possibility. As with all benchmarking studies, there is intrinsic value in re-implementing algorithms and re-doing studies. For example, our study does not show that classifier systems are necessarily worthless at playing strategy games; it is completely possible that there is a simple way (which we overlooked) of overcoming the problems we faced with the XCS agents and achieve a much better score. Still, it is interesting that despite our best efforts, we could not bring the XCS agent to learn strategies that outperformed even the simplistic SemiRandom agent.

The only two agent architectures that could reliably outperform all the benchmark agents (even in terms of win/loss ratio) were both tree search-based: MinMax and MCTS. This could be seen as a victory of game-tree search over non-searching methods. It is important to note that both the MinMax and MCTS agents used the same board evaluation function, which is a neural network trained by NEAT. (The board evaluation function was re-trained for each model.) Using the same evaluation function could explain the similar (but not identical) performance profile. Thus, the MCTS agent is just as reliant on a good evaluation function as the MinMax agent, so these results could as well indicate the superiority of neuroevolutionary state evaluation functions. The NEAT agent, which learns state-action mappings rather than state-value mappings, is among the better performing agents but scores slightly lower than the tree-search agent. This finding agrees with the findings of previous comparisons of state-value and state-action mappings in other games, such as racing games, where state-value mapping turned out to be superior [46].

The assigner framework, which was based on the idea of decomposing the MultiAction selection task hierarchically, did not meet our expectations. Both agent architectures that used this framework (XCSA and NEATA) performed remarkably worse than their counterparts which did not use the assigner (XCS and NEAT). We do not think this should be interpreted as evidence against hierarchical agent architectures in strategy games, but rather that it is hard to find the correct decomposition of a control/game-playing task. Some experiments in evolutionary robotics suggest that a decomposing a task manually might actually make it harder to learn, whereas allowing an evolutionary process to decompose the task and structure the neural network that learns it could bring significant improvements to performance [47]. This is taken as the basis for an argument that tasks should be decomposed (and controllers structured) from a “proximal” itself (the controller itself) rather than a “distal” perspective

(the human experimenter). Future work could involve creating agents that automatically learn assigner-like controller/task decompositions while they learn policies.

The relation between performance against the benchmark controllers and performance and perceived qualities when playing against humans deserves further comments. Let us start with win rates. It can be observed that controllers which won often against the benchmark controllers also won often against humans. We see that as a validation of the relevance of our benchmark agents. In fact, human players seem to be only slightly harder to win against than the FSM agent for most of the agents. On the other hand, the FSM agent plays very badly against humans, only winning 20% of games.

One significant result is that the NEAT agent is the agent that wins most often against human players (by a respectable margin) while being equally good or slightly worse than the tree search-based agents against benchmark agents. The NEAT agent also has a much lower win rate than win-loss rate against the SemiRandom agent, indicating that many games between NEAT and the SemiRandom agent tends to end in draws. On the other hand, the NEAT agent has a *higher* win-loss rate than win rate against the FSM agent. To gain more information about this interesting pattern, we visualised several games with the NEAT agent against both benchmark agents and humans in our game player. We observed that the NEAT agent had evolved a very conservative but effective strategy where it groups its agents close together, waits for the enemy to approach, and defends. This is a highly efficient strategy against aggressive players like the FSM agent (and most humans), but will most likely lead to a draw against a more passive player like the SemiRandom agent.

A take-home message of the above analyses (and similar analyses that could be done of the other agents) is that dominance relations between the agents here are non-transitive; a strategy that works against strategy A might not work so well against strategy B, even though A works well against B. In board game research, playing strength is often seen as a scalar property, i.e. a complete ordering. It is unknown whether this is true for games as complex as those models that were used in this study as well, and whether the observed intransitivity is just a sign that the learned strategies are unsophisticated.

At this point, we wish we would have been able to compare the outcomes of our experiments with the results in literature. Unfortunately, we are not aware of any study that investigates the capability of the same algorithm to play more than one strategy game, not of any study that compares a number of significantly different algorithms for playing a strategy game. The closest we can find is evidence collected in a recent survey of MCTS outperforming rival tree search algorithms in a number of domains which are not strategy games as defined here [48]. This is consistent with the good results achieved by the MCTS agent in our experiments.

Our human test subjects' perceptions of the agents' playing strength differ substantially from the actual win rate. For example, the MinMax agent and NEAT agent were perceived as equally challenging; yet, the MinMax agent won much less often, and was perceived as much more enjoyable and dramatically more human-like. Visual inspection of the playing style of the MinMax agent shows that it plays much more aggressively than the NEAT agent, and that while playing well, it tends to take

unnecessary risks which allow the human to win. In our opinion, it is quite clear that the MinMax agent would be better suited than e.g. the NEAT agent be included as a computer opponent in a commercial strategy game.

One somewhat depressing result for researchers eager to include sophisticated AI in games is that the agent which was deemed most enjoyable was the FSM agent, which was a pushover in terms of performance and also rated as the least human-like. It could however be argued that the results would have been different if the players were more familiar with the particular games being tested.

During development of the agents, no importance was given to their human-likeness. One future research direction that could be pursued could be to train the trainable agents to display human-like playing styles. This could be done either directly, using supervised learning to imitate logged human strategies, or indirectly, rewarding agents for displaying human-like behavioural traits [49].

All of the agents that we used were adapted from agent architectures used for other game-related problems. Part of the goal of the current project was to investigate how well these approaches fare when adapted to strategy games. We were quite surprised to see how well the NEAT agent fared, even though it only does a one-ply search. While the MCTS agent performed slightly better than the other agents overall, the benefits of all this search are quite slim compared to the much increased computation time. For a more complex strategy game (e.g. Civilization), even the current value-based MCTS would likely be too computationally expensive. In sum, the methods that are more often used for action games (NEAT, FSM and PF) worked surprisingly well, and the search-based methods traditionally used for board games performed less well than expected, probably because of the huge branching factor. This points to the need for more research on methods that can handle this branching factor.

10.8 Conclusions

Seven different agent architectures were implemented, trained and tested against three simple benchmark agents on six different turn-based strategy games implemented in the SGDL environment. The agent architectures are inspired by methods that have proven effective at playing board games, action games or real-time strategy games. Additionally, four of the agents and one of the benchmark agents were tested against human players, and the human players were surveyed for their perceptions of the agent.

It was found that a combination of game-tree search (either MinMax or a version of MCTS) and evaluation functions learned with neuroevolution performed best on average against the benchmark agents, while the NEAT agent performed best against human players but was rated as the least enjoying. A number of secondary findings, discussed in the previous section, will inform further research on developing strategy game playing agents that are capable of playing well, being human-like and/or fun to play against. The agents developed during this study are already being used as part of simulation-based fitness functions for the evolution of new game rules and game maps.

REFERENCES

1. J. L. Nielsen and B. F. Jensen, "Artificial agents for the strategy game description language," Master's thesis, IT University of Copenhagen, 2011. [Online]. Available: http://game.itu.dk/sgdl
2. T. Mahlmann, "Modelling and generating strategy games mechanics," Ph.D. dissertation, ITU Copenhagen, December 2012.
3. R. F. Nohr and S. Wiemer, "Strategie spielen," in *Strategie Spielen*. Lit Verlag Berlin, 2008, pp. 7–27.
4. S. Deterding, "Wohnzimmerkriege," in *Strategie Spielen*. Lit Verlag Berlin, 2008, pp. 29–68.
5. R. Reichert, "Goverment-games und gouvertainment," in *Strategie Spielen*. Lit Verlag Berlin, 2008, pp. 189–212.
6. R. F. Nohr, "Krieg auf dem Fussboden," in *Strategie Spielen*. Lit Verlag Berlin, 2008, pp. 29–68.
7. A. Turing, "Digital computers applied to games," in *Faster Than Thought (ed. B. V. Bowden)*. London, United Kingdom: Pitman Publishing, 1953, pp. 286–295.
8. A. Kotok and J. McCarthy, "A chess playing program for the IBM 7090 computer," Master's thesis, Massachusetts Institute of Technology. Dept. of Electrical Engineering, 1962.
9. A. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal*, vol. 3, no. 3, pp. 210–229, 1959.

10. M. Newborn, *Kasparov Vs. Deep Blue: Computer Chess Comes of Age*. Springer, 1997.

11. C.-S. Lee, M.-H. Wang, G. Chaslot, J.-B. Hoock, A. Rimmel, O. Teytaud, S.-R. Tsai, S.-C. Hsu, and T.-P. Hong, "The Computational Intelligence of MoGo Revealed in Taiwan's Computer Go Tournaments," *IEEE Transactions on Computational Intelligence and AI in games*, vol. 1, no. 1, pp. 73–89, 2009.

12. I. Szita, G. Chaslot, and P. Spronck, "Monte-Carlo Tree Search in Settlers of Catan," in *Advances in Computer Games*, 2009, pp. 21–32.

13. J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation: a taxonomy and survey," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. in print, pp. 172 – 186, 2011.

14. J. Togelius, M. Preuss, N. Beume, S. Wessing, J. Hagelbäck, and G. N. Yannakakis, "Multiobjective exploration of the starcraft map space," in *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG)*, 2010, pp. 265–272.

15. T. Mahlmann, J. Togelius, and G. Yannakakis, "Towards procedural strategy game generation: Evolving complementary unit types," *Applications of Evolutionary Computation*, vol. 6624, pp. 93–102, 2011.

16. ——, "Modelling and evaluation of complex scenarios with the strategy game description language," in *Proceedings of the Conference for Computational Intelligence (CIG) 2011*, Seoul, KR, 2011.

17. M. Buro, "Orts: A hack-free rts game environment," in *Proceedings of the Third International Conference on Computers and Games*, 2003, pp. 156–161.

18. J. E. Laird, A. Newell, and P. S. Rosenbloom, "SOAR: an architecture for general intelligence," *Artif. Intell.*, vol. 33, pp. 1–64, September 1987.

19. S. Wintermute, J. Z. Xu, and J. E. Laird, "Sorts: A human-level approach to real-time strategy ai," in *Proceedings of Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 2007, pp. 55–60.

20. J. Hagelbäck and S. J. Johansson, "A multiagent potential field-based bot for real-time strategy games," *Int. J. Comput. Games Technol.*, vol. 2009, pp. 4:1–4:10, January 2009.

21. D. Churchill, A. Saffidine, and M. Buro, "Fast heuristic search for rts game combat scenarios," in *Proceedinges of Artificial Intelligence and Interactive Digital Entertainment*, 2012.

22. R.-K. Balla and A. Fern, "Uct for tactical assault planning in real-time strategy games," in *Proceedings of the International Joint Conference on Artifical intelligence (IJCAI)*, 2009, pp. 40–45.

23. S. R. K. Branavan, D. Silver, and R. Barzilay, "Non-linear monte-carlo search in civilization ii," in *Proceedings of the International Joint Conference on Artifical intelligence (IJCAI)*, 2011.

24. J. K. Olesen, G. N. Yannakakis, and J. Hallam, "Real-time challenge balance in an rts game using rtneat," in *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, Perth, December 2008, pp. 87–94.

25. B. Weber, P. Mawhorter, M. Mateas, and A. Jhala, "Reactive planning idioms for multi-scale game AI," in *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, aug. 2010, pp. 115 –122.

26. N. Love, T. Hinrichs, D. Haley, E. Schkufza, and M. Genesereth, "General Game Playing: Game Description Language Specification," 2008.
27. C. Browne, *Evolutionary Game Design*. Springer, 2011.
28. M. Sicart, "Defining game mechanics," *Game Studies*, vol. 8, no. 2, pp. 1–14, 2008. [Online]. Available: http://gamestudies.org/0802/articles/sicart
29. K. Salen and E. Zimmerman, *Rules of play: Game design fundamentals*. Boston: MIT Press, 2003.
30. T. Mahlmann, J. Togelius, and G. N. Yannakakis, "Spicing up map generation," in *Proceedings of the 2012 European conference on Applications of Evolutionary Computation*, ser. EvoApplications'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 224–233.
31. S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.
32. K. O. Stanley and R. Miikkulainen, "Evolving Neural Networks through Augmenting Topologies," *Evolutionary Computation*, vol. 10, pp. 99–127, 2002.
33. K. H. Rosen, *Discrete Mathematics and Its Applications*, 5th ed. McGraw-Hill Higher Education, 2002.
34. M. Mitchell and S. Forrest, "Genetic algorithms and artificial life," *Artificial Intelligence*, vol. 1, no. 3, pp. 267–289, 1994.
35. J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. The MIT Press, Apr. 1992.
36. R. Rojas, *Neural Networks: A Systematic Introduction*. Springer-Verlag, 1996.
37. L. Kocsis and C. Szepesvri, "Bandit based monte-carlo planning," in *Machine Learning: ECML 2006*, ser. Lecture Notes in Computer Science, J. Frnkranz, T. Scheffer, and M. Spiliopoulou, Eds. Springer Berlin / Heidelberg, 2006, vol. 4212, pp. 282–293.
38. G. Chaslot, M. Winands, J. H. van den Herik, J. Uiterwijk, and B. Bouzy, "Progressive Strategies for Monte-Carlo Tree Search," in *Joint Conference on Information Sciences, Salt Lake City 2007, Heuristic Search and Computer Game Playing Session*, 2007.
39. S. Gelly and D. Silver, "Combining online and offline knowledge in UCT," in *ICML '07: Proceedings of the 24th international conference on Machine learning*. New York, NY, USA: ACM, 2007, pp. 273–280.
40. ——, "Monte-carlo tree search and rapid action value estimation in computer go," *Artif. Intell.*, vol. 175, pp. 1856–1875, July 2011.
41. P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine Learning*, vol. 47, pp. 235–256, 2002.
42. J. Hagelbäck and S. J. Johansson, "Using multi-agent potential fields in real-time strategy games," in *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems - Volume 2*, ser. AAMAS '08. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2008, pp. 631–638.
43. ——, "The Rise of Potential Fields in Real Time Strategy Bots," in *Proceedings of Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 2008.
44. J. Hagelbck and S. Johansson, "A multi-agent potential field based bot for a full rts game scenario," *International Journal of Computer Games Technology*, vol. 2009, pp. 1–10, 2009.

45. S. W. Wilson, “Generalization in the XCS Classifier System,” in *Genetic Programming 1998: Proceedings of the Third Annual Conference*, 1998, pp. 665–674.

46. S. M. Lucas and J. Togelius, “Point-to-point car racing: an initial study of evolution versus temporal difference learning,” in *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2007.

47. R. Calabretta, S. Nolfi, D. Parisi, and G. P. Wagner, “Duplication of modules facilitates functional specialization,” *Artificial Life*, vol. 6, pp. 69–84, 2000.

48. C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, “A survey of monte carlo tree search methods,” *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 4, no. 1, pp. 1–43, 2012.

49. N. van Hoorn, J. Togelius, D. Wierstra, and J. Schmidhuber, “Robust player imitation using multiobjective imitation,” in *Proceedings of the Congress on Evolutionary Computation*, 2009.