NTNU – Trondheim
Norwegian University of
Science and Technology

Opponent Modeling and Strategic Reasoning in the Real-time Strategy Game Starcraft

Magnus Sellereite Fjell
Stian Veum Møllersen

Master of Science in Computer Science
Submission date: June 2012
Supervisor: Helge Langseth, IDI
Co-supervisor: Anders Kofod-Petersen, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

Since the release of BWAPI in 2009, StarCraft has taken the position as the leading platform for research in artificial intelligence in real-time strategy games. With competitions being held annually at AIIDE and CIG, there is much prestige in having an agent compete and do well. This thesis is aimed at presenting a model for doing opponent modeling and strategic reasoning in StarCraft.

We present a method for constructing a model based on strategies, on the form of build orders, learned from expert demonstrations. This model is aimed at recognizing the strategy of the opponent and selecting a strategy that is capable of countering the recognized strategy. The method puts weight on the ordering and timing of buildings in order to do advanced recognition.

Sammendrag

Etter at BWAPI ble utgitt i 2009 har StarCraft tatt posisjonen som den ledende plattformen for forskning på kunstig intelligens i sanntids strategispill. Med årlige konkurranser på konferanser som AIIDE og CIG ligger det mye prestisje i å delta med en agent og å gjøre det bra. Denne masteroppgaven tar sikte på å presentere en modell for å gjøre fiendemodellering og strategisk resonnering i StarCraft.

Vi presenterer en metode for å konstruere en modell basert på strategier, i form av bygningsordre, lært fra ekspertdemonstrasjoner. Denne modellen tar sikte på å kunne gjenkjenne strategien til en motstander og velge en strategi som kan motvirke motstanderens strategi. Denne metoden legger vekt på rekkefølgen og tidspunktet til bygninger for å kunne gjøre avansert gjenkjenning.

Acknowledgements

This master thesis is the result of a project done at the Group for Intelligent Systems, Department of Computer and Information Science, The Norwegian University of Science and Technology.

We would like to thank our supervisors Helge Langseth and Anders Kofod-Petersen for their supervision and help during this project. We would also like to thank Gabriel Synnaeve for answering our questions regarding his research.

Our thanks also go out to the other guys at the StarCraft-lab for useful discussions and a good atmosphere throughout the semester.

Magnus Sellereite Fjell and Stian Veum Møllersen

Trondheim, June 4, 2012

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter introduces our master thesis. Section 1.1 gives an overview of the setting of our project, and the motivation behind it. Section 1.2 states our goals for this project. Section 1.3 states how we will test to see if our goals have been met. Finally, Section 1.4 provides an outline for the rest of our report.

1.1 Setting and Motivation

Opponent modeling and strategic reasoning are two central problems in artificial intelligence (AI). Opponent modeling is the problem of finding a way of describing an opponent in an adversarial game. This can be thought of as a classification problem, where the AI tries to map a set of observations to an abstract description of the opponent. Strategic reasoning is the problem of deliberating and selecting actions in a strategic manner. Strategic reasoning is closely related to opponent modeling, in an adversarial domain, by having a good description of the opponent an AI can reason more accurately and select actions better suited to counter whatever the opponent is doing.

Using real-time strategy (RTS) games for AI research has become increasingly popular in recent years. The strategy and real-time aspects makes these types of games interesting for the military industry[8], and with the growing entertainment industry the need for more fun and challenging games increases. As much as money motivates research, RTS games also have traits that make them interesting for AI researchers[8].

Recently, with the release of BWAPI[1], StarCraft with its expansion Brood War has become the leading RTS game for AI researchers[11]. The popularity of StarCraft

[1]`http://code.google.com/p/bwapi/`

has made it into one of the best selling RTS games of all time[40] and earned StarCraft a reputation as the Chess of RTS games. StarCraft also has a large competitive scene, which still has a considerable fan-base in South-Korea[7], that has generated much useful information about the game that can be used in research projects. BWAPI enables injection of code straight into the StarCraft-process which gives freedom to the programmer while keeping the original game-engine, instead of using a clone such as Wargus for Warcraft 2. Competitions, featuring agents made by researchers from different universities and hobbyists, are taking place at large conferences such as AIIDE and CIG contributing to the continued growth of StarCraft as an AI research platform.

We want to try to find a simple method of opponent modeling and strategic reasoning that authors of bots can use. With the increased interest in StarCraft bots, more and more non-academics are finding their way into the BWAPI community, trying to make good bots. Having a method that does not require in depth mathematical knowledge and that builds upon an intuitive way of understanding the opponent and the selection of build order gives an alternative to these people for opponent modeling and strategic reasoning. We also want our method to be free from requiring a deep knowledge of StarCraft, as many academics and researchers lack this kind of knowledge, having their strength in theoretical AI. Providing a method that can be improved both from mathematical knowledge and in depth StarCraft knowledge provides something for both types of StarCraft bot authors.

1.2 Goals

In this thesis we present the theoretical foundation for and implement a model capable of performing opponent modeling and plan selection applied to the real-time strategy game StarCraft. The goals of this model are:

1. Be able to to recognize the build order of an opponent.
2. Be able to select a plan, in the form of a build order, that counters the recognized build order of the opponent.

1.3 Experimental Setup

To support our conclusions and decide whether the goals have been reached, experiments will be conducted. The experiments will be centered around answering two questions:

1. Is the opponent model able to successfully recognize the build order of the opponent?
2. Is the counter plan selection model selecting viable counter plans?

These questions will be answered with two cross-validation tests. The first test will determine how well our opponent model is able to recognize the actions of the opponent and how well the model handles incomplete information. The second test will determine how well the plans of our counter plan selection model matches the plans created by skilled players.

1.4 Report Outline

- Chapter 2: **Background**
  This chapter contains the background material of our thesis. The chapter covers some relevant advanced strategical topics in StarCraft. In addition, a look on the theory behind opponent modeling, strategic reasoning and learning build orders is presented.
- Chapter 3: **Related Work**
  This chapter gives an overview of the related work conducted in the fields of opponent modeling and strategic reasoning, with special focus on work conducted in relation to real-time strategy games and StarCraft in particular.
- Chapter 4: **Methodology**
  This chapter presents our solution to opponent modeling and strategic reasoning in the context of StarCraft on a theoretical level. In addition, the chapter presents our solution to how we can learn the necessary data needed to realize this solution.
- Chapter 5: **Implementation**
  This chapter covers the tools used and work done to implement the theoretical solution presented in Chapter 4.
- Chapter 6: **Experiments**
  This chapter gives an overview of the experiments conducted on the models and presents the results of the experiments.
- Chapter 7: **Discussion**
  This chapter discusses the results of the experiments presented in Chapter 6 with regard to the goals of the thesis.
- Chapter 8: **Conclusion and Further Work**
  This chapter gives a conclusion to our thesis and presents ideas for further work.

- Appendix A: **Results**
  This appendix contains the raw numbers obtained from our experiments.
- Appendix B: **Graphs**
  This appendix contains the full size graphs created to present the results.

Chapter 2

Background

This chapter presents the background for our thesis. Section 2.1 gives a brief introduction to StarCraft, and the aspects that are relevant to our thesis. Section 2.2 gives an overview of opponent modeling in the context of RTS games. Section 2.3 gives an overview of strategic reasoning in the context of RTS games. Finally, Section 2.4 presents the problem of learning build orders and how it relates to StarCraft.

2.1 StarCraft

In this section we discuss some advanced strategic topics from StarCraft, as the general concepts of StarCraft were covered in our fall project, it will not be repeated in this section. We look at build orders in-depth and how they affect decision making of players. We discuss the flow of a game, how it progresses from the early game, through mid game, to late game, what characterizes the different phases of the game and important events that occur throughout a game. We also look at how these two topics relate to each other.

A build order in StarCraft is a term used to describe the order in which the player constructs his or her buildings during a game. Buildings in StarCraft enables the production of additional units and technologies, this is what progresses the game, from its early beginnings with only the central building and a handful of worker units, to the late stages of the game with production and research buildings, multiple bases and a large army of advanced military units.

Because the balance of power is not maintained at unit-level but rather at army-level, composing an army is not straight forward. Some units have a relationship akin to that of rock-paper-scissors, so pitting an army of tanks, a unit that can only

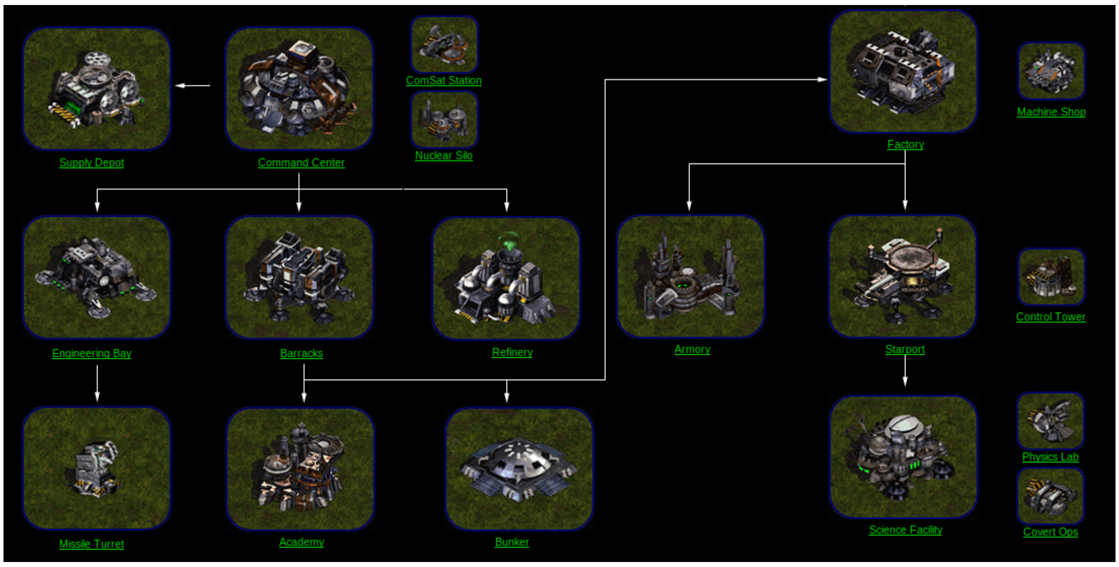


Figure 2.1: Illustration of the Terran tech tree.

target ground units, against an army of mutalisks, a flying unit capable of targeting ground units, is a one-sided affair. Every unit in StarCraft has a weak, exploitable side, knowing which unit counters which unit is a key piece of knowledge needed for successful play. Disregarding the mechanical skill of the player and positional benefits, one could make the argument that battles are largely decided by the composition of the armies. Since buildings enable the production of units, the order in which they are constructed decides when a player has access to which units, and when a player can potentially use a unit's strength to exploit a weakness in another player's unit. This is why buildings are one of the key strategical concerns in StarCraft.

Constructing buildings in StarCraft takes resources and resources are limited. Deciding which building to spend resources on and when to spend them is a key strategic decision every player is faced with when playing StarCraft. In a match between a Terran player and a Zerg player, going for an early Barracks as a Terran player enables Marines to be produced early, but will hamper your economic growth by using resources that could have been spent on more workers. Going for the delayed Barracks will ensure a healthy economy early. Say that the Zerg player is going for an early Spawning Pool and a Zergling rush. In the first case, the Terran player will most likely be able to fend off the early Zerglings with his Marines, but in the second case the Terran player will not have a sufficient number of military units to successfully defend his base. This simple example illustrates the importance of strategic decision making with regards to buildings.

Build orders are one of the main ways of relaying and discussing strategies in the StarCraft community. As a result of this, and years of meta-game development, most players follow a small, in comparison to all possible orderings of buildings, set of strategies. An example of this is a collection of Terran vs Protoss build orders found at Liquipedia[1]. This and the fact that build orders must adhere to the ordering of the tech tree, meaning that buildings can have other buildings as prerequisites, enables the player to guess or infer what the opponent is doing behind the fog-of-war. If a player observes that the opposing Terran has a Factory, a Barracks can be inferred because Barracks is a prerequisite of Factory, as illustrated in Figure 2.1.

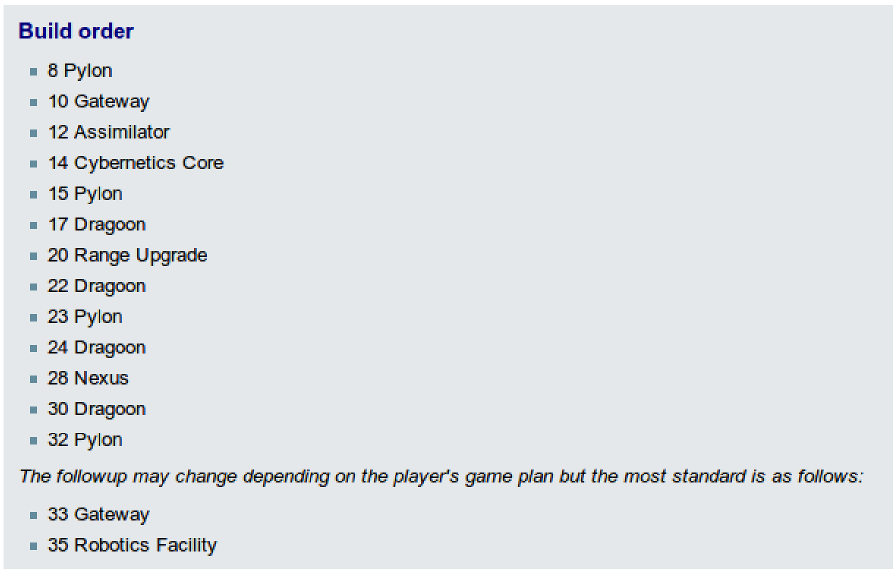
Time and timing is another important strategical aspect of StarCraft. A consequence of the way construction of buildings is limited by different factors, prioritizing which buildings need to be constructed when is a deciding factor for successful play. Connecting observations with the time of the observation can give a player important insight into the strategy of the opponent. Observing an early Command Center when playing against a Terran, a building that is a heavy investment with its high resource cost, enables the player to infer that the Terran has not invested heavily in military. This leaves the Terran vulnerable to early aggression or lets the player safely build economy instead of military.

Time and timing has different impact in different phases of a game. Early in a game a player has fewer possible choices to make, making it easier to play in an optimal manner. In the beginning of a game less information is also available about the opposition. This is reflected in the notion of openers, which are established strategies which are aimed at giving a player the best possible foundation. Openers have a high impact on the game play in later stages of a match. An aggressive opener can leave a player vulnerable if the opponent is allowed to catch up. Conversely, a passive opener can give the player the advantage by having a stronger economy later in the game. Exploiting weaknesses related to different phases of a strategy is something known as timing attacks.

As the number of variables impacting game play increases as the game progresses, the ability to read much from time becomes increasingly hard. Actions in this phase are more often reactions to events rather than planned. Small variances in play adds up over time, and the structure and predictability of build orders dissipates, making it harder to infer knowledge based on time and observations.

An example of a build order, as they are most often found in written information, can be seen in Figure 2.2. Time in build orders are on the form of worker count, this is to make it easier for humans to relate to, as game time is not easily accessible for human players since it is not displayed. A hallmark of optimal play is the continuous production of worker units, making this an easier metric to use for humans. Most professional level games of StarCraft is played with the speed-setting set to fastest, this means that 1 second in the game does not translate to 1 second

[1]`http://wiki.teamliquid.net/starcraft/Main_Page`



Build order

- 8 Pylon
- 10 Gateway
- 12 Assimilator
- 14 Cybernetics Core
- 15 Pylon
- 17 Dragoon
- 20 Range Upgrade
- 22 Dragoon
- 23 Pylon
- 24 Dragoon
- 28 Nexus
- 30 Dragoon
- 32 Pylon

The followup may change depending on the player's game plan but the most standard is as follows:

- 33 Gateway
- 35 Robotics Facility

Figure 2.2: Example of a textual build order. This particular example is of a fast expand Protoss vs Terran.

in real time. Using BWAPI as the interface circumvents this completely by tracking time in frames, which is independent of the speed-setting.

2.2 Opponent Modeling

Opponent modeling is mentioned as one of the key problems in RTS games[8]. The models are abstract descriptions of the strategic actions of the opposing player[31] and are used to do inference or reasoning in cases where information is scarce or missing from the player's point of view. Opponent modeling problems can often be thought of as classification problems, where the model is presented with a set of observations which is then mapped to a specific state. This mapping is done by a function that is either learned from examples or constructed by the programmer.

Opponent modeling can be done on several aspects of an RTS game, in our case we want to be able to recognize the build order of an opponent in a game of StarCraft. The observations are the buildings the player has observed from his opponent and the states are made up of learned build orders. Once a build order has been recognized we can perform lookahead from the current point in time and assuming the player continues to follow the recognized build order we can make a prediction about what comes next.

2.3 Strategic Reasoning

Strategic reasoning is the problem of deliberating and taking decisions in a strategic manner. Strategic reasoning is one of the big challenges in game AI, and RTS games in particular[8]. In RTS games, and other adversarial games, the main challenge is beating the opposing player or players. In this context strategic reasoning is used to devise a sequence of actions that will achieve objectives throughout the game and ultimately succeed in defeating the opponent. This differs from classic AI planning in the sense that the agent has to take into account the actions of the adversary.

For our thesis, this problem would translate to finding a good build order for an agent to follow in a game of StarCraft. A good build order would constitute a build order that has taken into account what the opposing force is doing, in other words a build order that would counter the build order of the opponent. This particular feature makes this problem tie nicely into the problem of developing a good opponent model, meaning that a good plan would be dependent on a solid understanding of the opponent's chosen plan.

2.4 Learning Build Orders

In order to develop a good opponent model we need to obtain knowledge of build orders used in StarCraft. There are primarily two ways to obtain this knowledge. The first option is constructing a set of predefined strategies, based on prior knowledge of strategies in StarCraft, which can be used to describe or identify build orders as a certain strategy. This way of describing build orders is referred to as labeling. The second option is learning build orders as they appear in the examples without dividing build orders into predefined groups.

One of the features that makes StarCraft a very suitable platform for AI research is the abundance of information in several different forms and formats. Having been around since 1998 and having had an unprecedented popularity for most of this period, professional players and enthusiasts alike have had a field day with theory crafting and play testing every inch of the game. This has resulted in a well-developed way of playing the game, which in many cases is bordering on the absolute optimal.

Information about StarCraft exists in mainly three forms:

1. Written information. This type of information is mostly the result of analysis or tests performed by a person or persons and then summarized in an article, wiki-entry or forum-post. The main drawback of this type of information is

that it often is condensed or summarized information susceptible to the bias of the writer(s) and it does not have a consistent form or format, making it hard to extract the useful information when learning.

2. Audible information. This is very similar to written information, it is most often found in the form of commentary tracks to matches or recorded analysis. This has the same drawbacks as with the written information, but it is even harder to extract useful knowledge when learning.

3. Visual information. This is perhaps the most knowledge-rich type of information about StarCraft, because it has not been condensed or exposed to bias. There are two primary forms of visual information about StarCraft. The first is recorded games, much like a normal video. This form is hard to use in learning, because visual representation is hard to interpret. The second form is replays, replays differ from recorded games because of the format and how the information is accessed. Replays are viewed with the game and for humans this does not differ much from normal recorded games, but for a computer this means that the format is consistent and accessible through the game itself. Replays contain all the information about a game, down to the specific actions performed by all players.

Large information repositories exists for all the types of information. Community sites like TeamLiquid[2] and GosuGamers[3] feature active forums and large data banks and unofficial leagues like iCCup[4] has large repositories of replays.

[2]`http://www.teamliquid.net/`
[3]`http://www.gosugamers.net/`
[4]`http://www.iccup.com/`

Chapter 3

Related Work

This chapter presents work that has been done that is similar and related to our thesis. Section 3.1 presents work that has been done with respect to opponent modeling. Section 3.2 presents work that has been done with respect to strategic reasoning.

3.1 Opponent Modeling

Our work is based on ideas presented by Weber in his paper on a data mining approach to strategy prediction[34], and Synnaeve and Bessière in their paper on a Bayesian model for plan prediction[29]. Weber presents an approach to opponent modeling using several machine learning algorithms applied on a collection of replays of StarCraft games in order to learn models that can be used to classify build orders. Synnaeve and Bessière presents a model based on a Bayesian network for classifying opponent build orders in StarCraft, using distributions with parameters learned from analyzing replays of StarCraft games. The main difference between the two approaches is the use of labels on replays. Weber uses labeled replays whereas Synnaeve uses unlabeled replays, as discussed in Section 2.4. Both Weber and Synnaeve have implemented their theories into agents, called EISBot and BroodwarBotQ respectively, that are capable of playing StarCraft at a high level[12, 16].

There are other related works in opponent modeling related to StarCraft. Hsieh and Sun[15] presents a player strategy model constructed with case-based reasoning techniques and replays of StarCraft games. The cases are represented as feature vectors and euclidean distance is used to find matching cases. Kabanza et al.[17] uses a technique known as hierarchical task networks (HTNs) to encode plans or strategies in StarCraft, and uses hidden Markov models (HMMs) to recognize plans.

This has been implemented in the StarCraft agent known as SPAR. Synnaeve and Bessière[28] used labeled replays to construct a Bayesian network capable of recognizing openers.

StarCraft as a test bed for AI research has only existed for the last couple of years, making related work with other, older RTS games relevant. Ontañón et al.[23] and Mishra, Ontañón and Ram[20] presents a case-based planning system enhanced with situation assessment to improve case retrieval. The situation assessment includes a model of the opponent, the map state, placement of troops and similar features. This is implemented in a system called Darmok2 for the game Wargus, a clone of Warcraft 2, the predecessor to StarCraft. Schadd, Bakkes and Spronck[27] presents a method of opponent modeling using hierarchically structured models. The hierarchy is constructed using hand authored models sorted by abstraction, where the higher levels describe more general behavior which gets more specific as it moves down through the hierarchy, tested in the RTS game SPRING. Chung and Buro[10] does evaluation of plans generated with Monte Carlo simulation by simulating execution against a plan generated by an opponent in OpenRTS.

Opponent modeling, and plan recognition in particular, is also a highly relevant problem in more classic adversarial games. Ramírez and Geffner[26] discuss plan recognition as a planning problem, generating a plan that explains a set of observations thereby identifying which constraints that have been satisfied in order to generate the plan. Fagan and Cunningham[13] successfully used case-based plan recognition to do player modeling in a Space Invaders clone called COMETS. Charniak and Goldman[9] presents Bayesian models as a way of doing plan recognition with uncertainty. Albrecht, Zukerman and Nicholson[3] uses dynamic Bayesian networks as a way of doing plan recognition in a multi-user dungeon.

3.2 Strategic Reasoning

Work in strategic reasoning, the act of deciding which action to take next, can be divided into two directions, planned and reactive. Reactive reasoning performs no look-ahead and maps the game state to appropriate actions or behavioral patterns in a strategic manner. Planning attempts to foresee future events and plan ahead in order to reach a goal. In complex domains, such as RTS game AI, planning faces challenges with plans becoming invalidated and constantly having to re-evaluate plans.

Weber et al.[36, 33] applies a reactive technique known as goal-driven autonomy to select actions in StarCraft, using ABL[18] to define a conceptual model that is able to pursue goals in a concurrent manner enabling it to react quickly to events. Goals are extracted from replays using a case-based formulation[35, 37] and retrieved from the case-base using a technique enhanced by conceptual neighborhoods[32]. This is all implemented in the StarCraft agent EISBot. Ontañón et al.[22] presents a method for learning plans to be used with Darmok2 from unlabeled replays. The method is based on extracting information from replays to form plan dependency graphs, graphs that detail the relationships among actions in a plan, based on a set of predefined goals to look for in the replay. Weber and Ontañón[39] further builds on this by presenting a method of automatically annotating replays according to a goal ontology, used in EISBot. This is further improved by Weber et al.[38] with a method of learning goals, expectations and explanations for goal driven autonomy directly from replays. Churchill and Buro[11] uses a set of heuristics and constraints in the form of dependencies and goals to optimize build order plans. This has been implemented in the StarCraft agent UAlbertaBot.

Related works in real-time strategic reasoning is found in other adversarial games, while the complexity is often lower than that of StarCraft, the work still holds relevance. Aha et al.[2] uses a case-based reasoning system, based on the task decomposition by Aamodt and Plaza[1], called Case-based Tactician (CaT) in Wargus. CaT uses a state lattice based on the buildings available, developed by Ponsen[24] to represent the game state in the cases. Ponsen et al.[25] further improves this state lattice by presenting an automatic state-based tactic generator based on evolutionary learning called ESTG. ESTG is capable of evolving knowledge bases consisting of state-based tactics by searching for counter-tactics to provided tactics with evolutionary algorithms. Molineaux et al.[21] uses a reinforcement learning/case-based reasoning-hybrid called Continuous Action and State Space Learner (CASSL) to do continuous action selection in MadRTS. CASSL's approach takes a less knowledge intense approach than CaT making it less reliant on a large state-space taxonomy and is more suited to deal with a large number of primitive actions because CASSL does not select actions from a set of predefined actions.

Balla and Fern[6] presents a system using UCT, a Monte Carlo planning algorithm, to do tactical assault plan generation in Wargus. The system was able to successfully generate good plans, compared to humans, for controlling units in battle. Synnaeve and Bessière[30] uses a Bayesian network to select actions for detailed unit control in StarCraft. This system is realized by letting each unit be controlled by an independent Bayesian network which receives commands as perceptions from a higher level module. Hoang, Lee-Urban and Muñoz-Avila[14] presents a system using Hierarchical Task Networks (HTNs) to encode strategies in the first person shooter Unreal Tournament. HTNs are used to reason about higher level goals and tasks, rather than actions on a primitive level.

Wintermute et al.[41] presents a cognitive system capable of playing ORTS based on the SOAR cognitive architecture called SORTS. SORTS is capable of creating agent behavior mimicking how humans reason, and supports the cognitive concepts of grouping and attention. SORTS did well in the ORTS competition at AIIDE

2006, where it won two out of three categories. Arrabales et al.[5, 4] presents CERA-Cranium, a general purpose test-bed for machine consciousness research which they interface with Unreal Tournament to control agents playing the game.

Chapter 4

Methodology

This chapter presents the methodology behind our thesis. Section 4.1 presents our solution to opponent modeling. Section 4.2 presents our solution to strategic reasoning. Section 4.3 presents our approach to aquiring the material needed to construct our models.

4.1 Opponent Modeling

In this section we discuss our solution to opponent modeling in Starcraft. We use ordered sequences of building-time pairs to represent states in a tree. Each unique sequence, meaning the unique order of the buildings, represents a state and transitions are handled by expanding the current state with a new building. The goal of this structure is to represent build orders in Starcraft in a way that best retains the information they contain and be able to handle the way games progress through different phases.

4.1.1 Constructing the Opponent Model

Our solution to opponent modeling using build orders in Starcraft is representing them as states in a tree. Each state has zero or more children and only one parent, and represents a unique sequence of buildings and the time which the state is most likely to be observed. Transition from one state to another is done by adding a building to the sequence, creating a new sequence. Since sequences are ordered, transition from one state to another will always result in moving to an immediate neighbor by following a link in the tree. The time in each state is represented using a Gaussian distribution, to account for discrepancies in timing across examples.

Since all build orders are supersets of another build order, all the way down to the starting structure, a path through the tree will represent a complete build order.

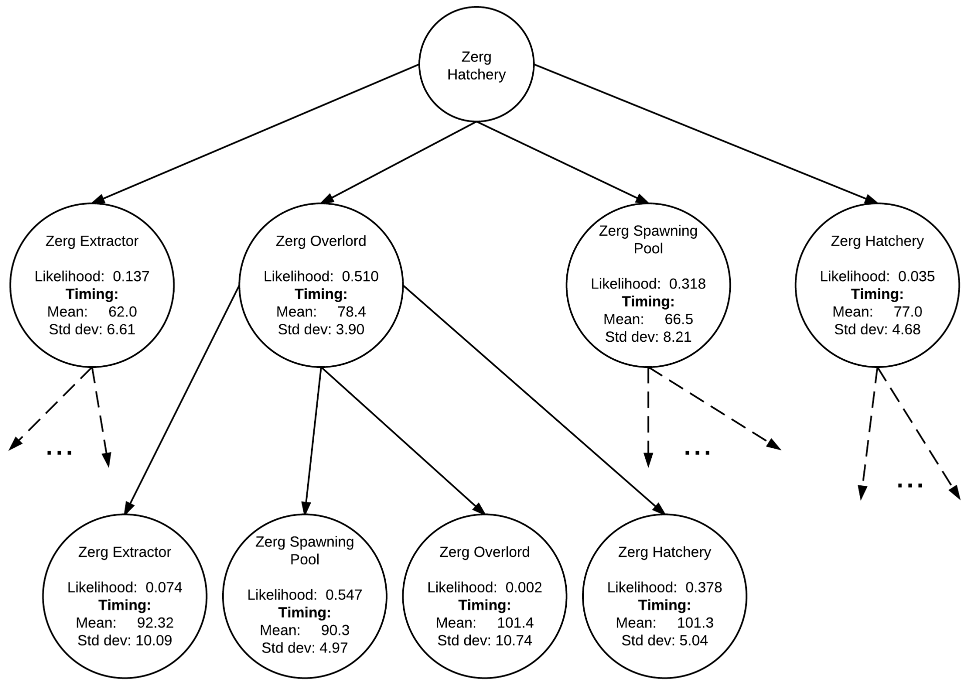


Figure 4.1: A section of the tree, illustrating the internal ordering of the states.

A section of the tree is illustrated in Figure 4.1. Here we see how each state is made up of the following parameters:

- Building:
  The building which separates this state from the previous state.
- Likelihood:
  The likelihood that this state follows from the previous state.
- Children:
  A reference to the children of this state.
- Mean:
  The mean of the Gaussian distribution representing the most likely time for the state.
- Standard deviation:
  The standard deviation of the Gaussian distribution.

For example, if the current state is {`Zerg Hatchery`}, the possible next transitions are its children, {`Zerg Hatchery, Zerg Extractor`}, {`Zerg Hatchery, Zerg Overlord`}, {`Zerg Hatchery, Zerg Spawning Pool`} and {`Zerg Hatchery, Zerg Hatchery`}, see Figure 4.1. If the state transitions to {`Zerg Hatchery, Zerg Overlord`}, it cannot transition to {`Zerg Hatchery, Zerg Extractor, Zerg Overlord`}, but only to the children of the {`Zerg Hatchery, Zerg Overlord`} state. This is because {`Zerg Hatchery, Zerg Overlord`} and {`Zerg Hatchery, Zerg Extractor`} are two distinct states with no common children.

The reason we represent our opponent model as a tree is that build orders evolve gradually while a game is played, meaning that all build orders can be seen as a superset of a shorter build order. Each time a building is constructed it defines a step in the build order. It is not possible to undo a building, even if it is destroyed it will still be a part of the build order for the rest of the game. This is exactly what a tree represents, the transition from a short build order to a larger one, following transitions from state to state creating a path in the tree.

In our model each transition, or link, represents the construction of a new building. This is a natural way of representing build orders. At each step in a build order you have to deliberate which building should be constructed next and as a build order progresses, it branches out and creates new potential paths. Theoretically there is an unlimited number of possible states, but in actual play there is a finite set of build orders that are actually in use. Since we are building the model based on observed examples, we do not have to account for this possibility.

Some states will only be observed few times, these are build orders that at one point have diverged from previously observed build orders. The build order up until the point of divergence has been observed in other examples, since build orders are supersets of shorter build orders. The information gathered from the example up until the point of divergence is transferable to other build orders that are supersets of the same partial build order. Following paths in the tree with few observed examples is avoided by having a measure of the likelihood of transitioning to a state.

When considering build orders, static defenses are not included as buildings. Static defenses are not part of a deliberate build order, but a reaction to the actions of the opposing player, usually an attack or imminent threat, much in the same way units are. The reasoning behind constructing a static defense structure is that it defends the player until the planned build order can be achieved and have an effect. Observing a static defense does not give any more information other than that the player is ready to meet an attack, and has to have slowed down his own build order in order to make the static defenses. As most static defenses have low to moderate resource cost and construction time, the impact on build orders from having to construct static defenses is represented within the standard deviation of the time distribution.

Supply buildings could be viewed in the same way as static defenses, a necessary evil needed to be able to realize the planned build order. Supply buildings have no direct effect on the technological level of a build order, however they can be used to say something about the size of the player's army, as supply is built as needed. This means that the number of supply buildings is a good estimate on how many units the opponent has. Another reason for including them is that they are a heavy investment early in the game, delaying more advanced technology both by their own resource cost, and the potential resource cost of the units they supply. Including supply buildings also makes it easier to differentiate between a fast technology rush, and a steady army push, as they contain many of the same buildings, only at different times.

As mentioned in Section 2.1, build orders are usually timed by worker count to ease execution for players. This is a good way to time building construction when players are following build orders for themselves, but when trying to predict which build order the opponent is following it is hard to count all his workers. Using worker count only hides timing using minutes and seconds behind something that is more easily understood by humans, but as an AI there is no problem keeping track of the time. Information about the game is given frame-by-frame, giving a fixed set of time for each call to the AI. This is an extremely fine-grained time resolution, and we scale it down to 1 time tick each 25 frames, which is roughly 1 second on the fastest speed setting. This is done as a finer resolution is not needed, and having a 1 second resolution makes it more intuitive for a developer.

When adding new examples to the tree, the construction time of a building will not decide whether or not it will be unified with a matching building. The time will, however, impact the Gaussian distribution by altering the parameters of that particular state. The main benefit of including time as Gaussian distributions is to ease the use of observations to recognize which state the game is currently in. The reason we use a Gaussian for the timing of a state is that buildings are not made at the exact same time for a build order, there are many different variables that affect the timing of a building. However, given enough examples the timing of a building will even out and fall within a time interval which we represent as a Gaussian. Since players strive to achieve the optimal timing for each building, those that follow the same build order will construct their buildings as close as possible to this optimal timing.

4.1.2 Using the Opponent Model

Using the opponent model to infer which build order the opponent is following is a matter of mapping a set of observations to a state which represents the build order, as discussed in Section 2.2. To do this in our opponent model we need to gather observations of the opposing player from the game. These observations are on the form `(the building observed, time when observation was made)`. Each time a new observation is obtained, a query will be made to the model with all the observations obtained in this game and the current time in the game. The model will attempt to map the observations to a specific state, giving the player information about which build order the opponent is following. By knowing which build order the opponent is following, we can predict his upcoming buildings, based on the assumption that the opponent will continue to follow the recognized build order until otherwise can be determined from an observation.

To recognize the state, our model will traverse the tree and mark a state as valid or invalid. The way this is done is by comparing the building and time of the observation, to the building identifying the state and the Gaussian distribution representing the time of the state. If the building in the observation matches the building identifying the state and the time of the observation is greater than the mean of the Gaussian distribution, that is, the average over the times it has occurred, the state is marked as valid. An observation can only validate one state per path. If we can form a continuous path from the root of the tree to a state within the current game time and the path encompasses all the observations, we have a potential recognition match. If we have several potential matches, we use the likelihood of each transition in a path to determine which state is the best match.

If we get no potential matches, we need to have something to fall back on. The fall backs are done in iterations:

1. Compare observation time to (mean - standard deviation) of the Gaussian distribution of a state.
2. Compare observation time to (mean - two standard deviations) of the Gaussian distribution of a state.
3. Compare observation time to (mean - three standard deviations) of the Gaussian distribution of a state.
4. Ignore time in observations and use only the buildings in the observations and the current game time.
5. Ignore observations and use the likelihood of each transition to output the most likely state that has a time within the current game time.

The reason time is used in this way when validating states is the fact that an observation of a building cannot happen before the building has been constructed. This means that a state occurring late in a game cannot be validated by an observation made early in the game. Observations are sorted with regard to time so that if there are two observations for the same building, the earliest observation is "consumed" first, to avoid problems with a later observation marking a node as valid, and then the earlier observation is not later than the next occurrence of the same building. This will limit the amount of paths considered for validity.

For example, given the observations {`(Protoss Pylon, 60), (Protoss Gateway, 64), (Protoss Pylon, 66)`}, and the state {`Protoss Gateway, 69, 4.31`}. Using only the mean as the time of the state, the observed Protoss Gateway does not fit this state. Using one standard deviation, the time of the state becomes `64.69`. The observed timing of the Protoss Gateway is `64`, and therefore not later than the timing of the state. Using two standard deviations, the timing of the state is `60.38`, and the state is then validated by the observation, leaving only {`(Protoss Pylon, 60), (Protoss Pylon, 66)`}.

If we have two states {`Protoss Pylon, 59, 2.14`} and {`Protoss Pylon, 65, 3.11`} we can see that both states will be validated by one observation each, as `60` is later than `59`, and `66` is later than `65`. If the observations were not sorted by time, but changed places, only one state would be validated by the observations. The state with a mean of `59` would be validated by the observation of a Protoss Pylon at time `66`, and leave one observation left at time `60`. The observation at time `60` would not validate the node with a mean of `65`, as that observation happened too early.

Given the selected path {`Protoss Pylon, 59, 2.14`}, {`Protoss Gateway, 69, 4.31`}, {`Protoss Pylon, 80, 3.11`}, {`Protoss Cybernetics Core, 90, 5.09`}, {`Protoss Forge, 107, 4.67`} found using one standard deviation, and the time 75. The recognized build order is {`Protoss Pylon, Protoss Gateway`} as the other states have a time that is later than 75. 1 lookahead gives a build order of {`Protoss Pylon, Protoss Gateway, Protoss Pylon`}. Predicting further ahead adds {`Protoss Cybernetics Core`} at 2 lookahead, and {`Protoss Forge`} at 3 lookahead.

4.1.3 Comparison of Related Works

As mentioned in Section 3.1, our work is inspired by Weber[34] and Synnaeve and Bessière[29]. We use the same material, build orders, to perform opponent modeling with the intent to predict actions.

The main differences between our approach and that of Weber is the use of predefined strategies as the target of classification and how he does inference. Instead of using predefined strategies we handle build orders as they appear in examples. This is done because representing build orders in the same way as Weber takes away some of the potential information in examples. Coercing build orders into categories like this can result in missing emerging trends or potentially new strong strategies. Where Weber uses different traditional classification algorithms, we use a searching algorithm based on depth first to search our model instead of classifying by comparing feature vectors. This is mostly a result of the different data structure used and follow as a consequence of the difference in how examples are treated.

Compared to the work of Synnaeve and Bessière the handling of examples is very similar, build orders are treated as they appear in examples and with the same notion that all build orders are supersets of shorter build orders. The main difference to this work is how inference is done. Synnaeve and Bessière uses a model based on Bayesian Networks to recognize the build order of the opponent. This approach has clear limitations in the way time is handled with observations. A "sanity check" is performed with observations, to ensure that only build orders that is a superset of the observations are considered for recognition. Observations, however, are not timed and therefore does not contain as much information as they could have. This is particularly a problem when build orders exceed a certain point in time. An early observation of a high tech building should indicate that the opponent is doing a fast tech, but if time on observations are omitted an early observation of a high tech building could potentially be used to indicate a slow tech build, just as common as fast tech, if the game has gone on for long enough. Adding this information to the Bayesian network presented by Synnaeve and Bessière would require the learning of $P(O_{i \in [[1 \ldots N]]}|T)$ for all possible observations, which would be intractable. In addition, observation vectors are harder to learn from replays and more volatile than build orders, meaning that patterns are rarely repeated from example to example, making it hard to get statistically significant numbers.

4.2 Strategic Reasoning

In this section we discuss our solution to strategic reasoning in Starcraft. We use a collection of trees that represent counter plans mapped to specific opponent build orders, represented by states in the opponent model. The goal of this structure is to do planning in Starcraft in a way that is highly flexible and reactive.

4.2.1 Constructing the Strategic Reasoning Model

Our solution to opponent modeling is also used for strategic planning. By learning a mapping between states of the opponent model to a model of counter plans, we can select plans for a player to follow. The way we do this is by finding all the build orders of the opponent that start with the partial build order represented by a state in the opponent model, and make a tree based on the build orders that are reactions to that build order. Since each replay consists of two build orders, one for each player in the game, we can use the build order of one side to build our opponent model and the other build order to build a counter plan selection model. These two build orders reflect the reactions each player has on the other player's build order, so we can use both sides for both purposes. The structure of the counter plan selection model and the mapping from the opponent model is illustrated in Figure 4.2.

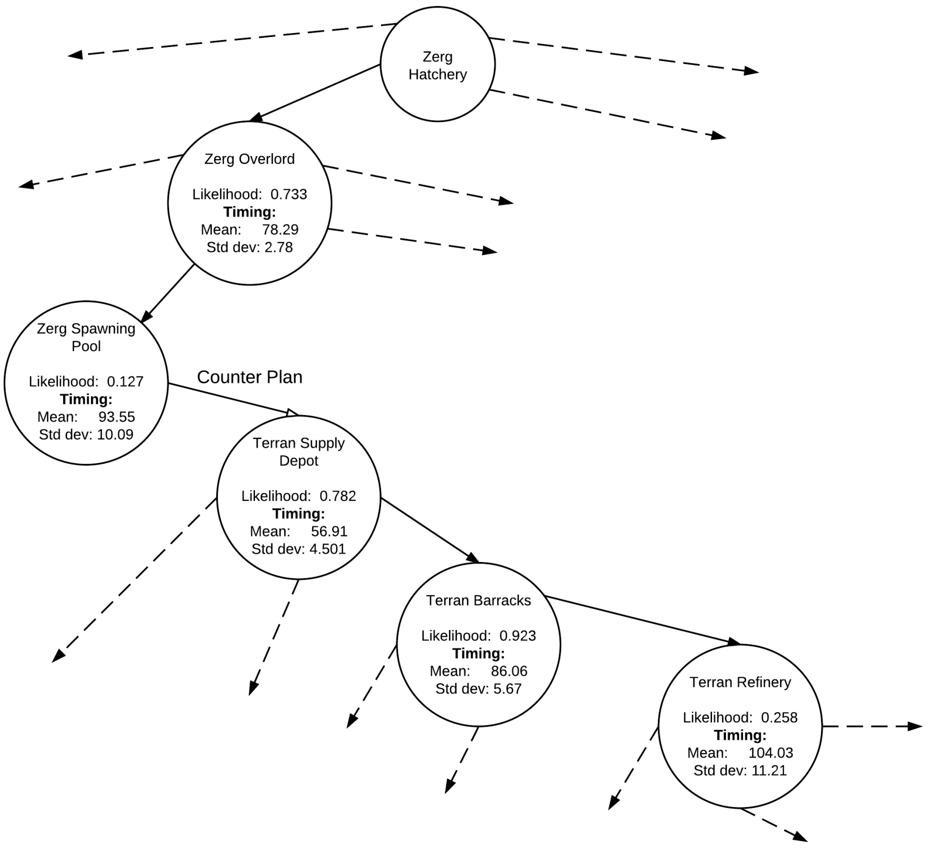


Figure 4.2: A section of the plan selection tree, illustrating the internal ordering of the states.

This model is based on the assumption that players, regardless of the outcome of the game, make well informed and sane actions in reaction to whatever the opponent does. This way we can justify the output plan from this model based on the theory that actions observed often from a group of examples from skilled players are good actions, while not differentiating between winning and losing build orders. The entire counter plan selection model is comprised of several separate trees, on the same form as the tree used in the opponent model, which has a mapping to a specific state in the opponent model tree.

The main motivation behind this way of planning is to make planning into an act of reaction, rather than deliberative, and as such is highly dependent on our opponent model being good. Counter plans are represented as build orders, so the same reasoning used for the structure in the opponent model is used for this structure.

4.2.2 Using the Strategic Reasoning Model

After recognizing the opponent build order with the opponent model, a counter plan can be obtained by following the mapping to the appropriate tree from the opponent model. To avoid changing build order after having invested resources, we select the the path that best fits our current build order. We use the same strategy choosing the best path in the counter plan tree, as in the opponent build order tree, only using our own build order as observations. We have full knowledge of our own build order, and to best fit the path, we first choose a path that fully fits our build order. If there is no such path, we use the same strategy as when finding the opponent build order, using the time of the building and the mean and standard deviations for each state in the counter tree.

Selecting which building to build next is a matter of finding the best path through the counter tree. The best path is decided by the frequency with which the counter plan is observed in the examples, in the same way likelihood decides in the opponent model. This opens up for re-planning during the game, as the counter is only selected based on a partial build order, and subsequent buildings from the opponent are assumed to follow the same path. As the game evolves, and more and more information is available, the amount of build orders that counter the opponent lessens, and the counters get more and more specific. Which path is the best depends on the cost of changing build order in order to adhere to the new one, thus making a path that suits your current build order the best. In effect this means that it is easier to change builds earlier in the game, than after you have dedicated yourself to a build.

4.3 Learning Build Orders

To create our opponent model and strategic reasoning model we need to obtain information about which build orders are used when playing and which build orders are used when playing against a specific build order. For the opponent model, observing a sufficiently large number of examples will provide us with a good model, based on the assumption that build orders are partially recurring.

In other words, that there exists build orders that are better than others giving players a reason to strive towards copying them. For the strategic reasoning model, observing examples of viable counter plans to specific build orders will provide us with a good model. Finding viable counter plans means observing skilled players, because skilled players make good and effective plans. This means that we need to obtain examples of highly skilled players to learn build orders from.

Several sources of information that are viable for learning build orders exists, but we have chosen to extract build orders from replays. Other sources consist of more abstract information, suited for human interpretation, such as written guides or analysis. Abstract information is hard to put to use in an AI, because the format varies greatly and will often require a semantic understanding on the level of a highly skilled Starcraft player to interpret. Abstract information is also often a condensation or generalization made over a large amount of examples, with no way to verify the process or ensure that no information is lost along the way. Replays however, offer highly detailed accounts of game on the form of atomic actions, which enable the game engine to perfectly recreate the game with full information. Because recording games is a feature built into the game and sharing replays is one of the primary forms of knowledge sharing in the Starcraft community, a large amount of replays can be found in various repositories.

The information contained in a replay is hard for a human to interpret, because of the resolution of the information. A single atomic action is nonsensical to a human, and the large amount of atomic actions needed to constitute useful information makes it infeasible for human use in its current form. This is why the game engine is used to recount the game for a human, on a level of abstraction that is meaningful. From an AI perspective however, replays in their raw form are very useful. Replays present the AI with information that can be quantified without semantic understanding of the domain. This enables us to automate the learning process and make use of the vast amount of examples available.

Being able to quantify data and create meaningful generalizations without being knowledgeable about the game means that the methods used to do this can be reused in domains who share the mechanic of build orders, without having to re-implement some semantic understanding of the domain. This is also the motivation behind choosing the approach of unlabeled examples, as opposed to the approach of labeled examples. A supervised learning method requires the examples to be labeled, which again require some form of abstraction or generalization based on semantic understanding of the game domain.

Weber[34] uses a set of labels based on common tactics in Starcraft games to label build orders. The benefit of this approach is that he is able to reason on a higher level of abstraction, but it may be susceptible to information loss in the process of labeling. And due to the way the meta-game in Starcraft could shift with the discovery of new strategies he could be forced to re-evaluate his labels. A learning method using unlabeled data, as Synnaeve and Bessière[29] does in their opponent model learning, would not be able to reason at the same level of abstraction, but it is able to adapt to changes in the strategies much more easily. Using unlabeled data also retains all the information contained in the data and is therefore not susceptible to losing information that could potentially be highly useful.

Chapter 5

Implementation

This chapter presents how we implemented our models. Section 5.1 describes the tools we have used. Section 5.2 presents how we construct our models.

5.1 Tools

To implement our theories we depend on a few tools to do some of the tasks required. We need a way to acquire raw materials, on the form of replays of games from good players, and a way to process these data into a format we can use. For this job we have decided to use Broodwar_replays_scrappers by Synnaeve[1] to acquire replays and bwrepanalysis by Fobbah[2] to process the replays.

5.1.1 Broodwar_replays_scrappers

Broodwar_replays_scrappers is a collection of Python scripts that downloads replays from know repositories of replays. The scrips require make, Python and pyreplib (optional, required to run verify and sort) to run and do three things. The scraper will download replays from GosuGamers[3], iCCup[4] and TeamLiquid[5]. The unifier compares the hashes of replays to eliminate duplicates. The verifier and sorter will attempt to verify all replays while trashing the corrupt ones and sort the replays under the correct match up.

[1]https://github.com/SnippyHolloW/Broodwar_replays_scrappers

[2]http://code.google.com/p/bwrepanalysis/

[3]http://www.gosugamers.net/starcraft/replays.php?

[4]http://www.iccup.com

[5]http://www.teamliquid.net/replay/index.php

5.1.2 bwrepanalysis

bwrepanalysis is a BWAPI module that extracts various information from Starcraft replays. It requires BWAPI, Chaoslauncher[6] and Starcraft: Brood War v. 1.16.1 to run, which is the same requirements as you have for running any kind of BWAPI module or bot. While analyzing a replay bwrepanalysis will output the following information:

- Replay Location Data (*.rld):
  The location of all units at a given frame number.
- Replay Game Data (*.rgd):
  All game events of a replay.
- Replay Order Data (*.rod):
  The orders given by all players during a game.

The information is saved in three different files as raw text, which is easy to interpret for a computer program. For our thesis we use the information in the *.rgd files.

The *.rgd Format

This is the format of the files that are the result of the replay game data analysis. These as simple texts files listing each piece of information on a separate line. Each file begins with some basic information about the replay. It lists which map the game is played on, the number of possible starting positions on the map and the names and races of each player that participated in the game. It then goes on to list the different events that happened in the replay:

- **Discovered:** A player's unit was discovered by the enemy.
- **Start/finish/cancel Research:** Events regarding research.
- **Start/finish/cancel Upgrade:** Events regarding upgrades.
- **Resources:** Each 25 frames the resources for each player is logged
- **PlayerLeftGame:** A player left the game
- **Nuclear Launch:** A nuclear missile has been launched
- **Unit Created:** A player's unit, no difference between units and buildings, has been created
- **Unit Destroyed:** A player's unit, no difference between units and buildings, has been destroyed

[6]`http://winner.cspsx.de/Starcraft/`

- **Morph Unit:** A Zerg unit or building has been created or upgraded from a lower tech unit/building, or a siege tank has changed mode to or from siege mode.
- **Changed Ownership:** A unit changed ownership, this happens when a player has left and lost control of their units

Each line is a separate event, all information regarding the event is contained on that line. Each event type has its own format on how and where that information is presented, but all start with the same three fields: `Frame#`, `Player ID`, and `Event`. The fields following these are dependent on the type of event, for instance a `Resource` event lists the current amount of gas and minerals and the total cumulative amount gathered during the entire game, while a `Unit Created` event lists the unit ID, unit name and the position of the unit.

If the replay is successfully analyzed, the file ends with an `[EndGame]` to signal that there were no errors, if this is missing, it means that the replay was broken in some way or another.

5.2 Implementation

The implementation process consists of three steps:

1. Acquire and process replays
2. Parse replays
3. Construct models

Step 1 rely on the use of external tools while Step 2 and 3 is the implementation of our theories.

5.2.1 Replay Acquisition and Processing

We want replays of good players, because we have based our design on the assumption that good players make good decisions. Using Broodwar_replays_scrappers we acquire replays from the "Top replays" section of GosuGamers, the "Gosu"[7] section of iCCup and TeamLiquid. The quality of replays from GosuGamers and iCCup is ensured, because of the categorization. The quality of replays from TeamLiquid is ensured because of the quality of TeamLiquid, which is the home of many professional and highly skilled hobby gamers. In addition to these sources we

[7]Gosu is a Korean term used to refer to a highly skilled person.

used replays gathered by Fobbah[8], to achieve a sufficiently high number of replays. The distribution of replays across match-ups can be found in Table 5.1.

Table 5.1: Distribution of replays by match-up.

Player race	**Opponent race**		
	P	**T**	**Z**
P	422	2152	3657
T	2152	422	3745
Z	3657	3745	1836

The replays are then processed with bwrepanalysis and the *.rgd files of the replays are collected.

5.2.2 Replay Parsing

We parse the all *.rgd files using a parsing script implemented in Python. We obtain `Unit Created` events and collect the unit-name and the time of the event. We then compare the unit-name to a list of the relevant building-names to make sure we only get the units (in a BWAPI context, a building is a type of unit) we want. This process is done for both players in the game. The build orders are then stored as a pair, since we want to use the data to create a model of reactions to build orders. The result of this process is a list of build order pairs, where each build order is an ordered sequence of buildings and times when the buildings were constructed.

5.2.3 Model Construction

The model construction is implemented as a Python program. The states in the tree are learned from the collection of build orders. There is one tree per race per match-up. The learning is done by collecting `(building, time, [previous buildings], next building)` from a build order and unify on `(building, [previous buildings])` to get a list of `(building, [previous buildings], [next buildings], [times], count)`. The parameters of the Gaussian distribution representing time is calculated based on `[times]`. We organize the data in a tree with links between states where a link is created between states $s1$ and $s2$ if $Building_{s2} \in NextBuildings_{s1}$. We calculate the likelihood of all the children of a state with Equation 5.1. The finished tree is a collection of states on the form of `(building, [previous buildings], [children], mean, std.dev., likelihood)`, where `[children]` are the links from the state to all

[8]`http://code.google.com/p/bwrepanalysis/`

child-states. This structure allows us to traverse it by following the links from parent state to child states.

$$\frac{Count_{child \in Children_s}}{\sum_{child \in Children_s} Count_{child}} \tag{5.1}$$

There is also a mapping from each state in the opponent model to the appropriate counter plans, which is implemented in the same way as the opponent model tree.

Chapter 6

Experiments

This chapter presents our experiments and our results. Section 6.1 presents the setup and results of our experiments for the opponent model. Section 6.2 presents the setup and results of our experiments regarding strategic reasoning. Section 6.3 presents our metric for evaluating our experiments.

6.1 Opponent Modeling

The main purpose of our opponent model is to be able to recognize the build order, or strategy, of the opponent, and to see if the model fulfills this purpose we need to test if the model is capable of recognizing build orders. The aim of this experiment is to gather some key numbers that will tell us how our model does in regard to this purpose.

6.1.1 Setup

For this experiment we perform a cross-validation test for each match-up, with an opponent model learned from 90% of the available material and tested with the remaining 10%. For mirror match-ups we use each replay two times, one time for each player perspective, which means we only need 5% of the original amount to construct a sufficient test set. From the test set replays we gather the build orders of the opponent, these serve as the control group. For observations we take the opponent's buildings that have been constructed thus far, shuffle them and take a portion, decided by the noise parameter, as our observations. 10% noise means that we remove 10% of the buildings, 20% means that we remove 20% of the buildings, etc. The time for observations are taken as the construction time of the building in the replay. We give the observations to the opponent model and let the opponent model recognize the build order from the observations. Once a build order has been recognized we compare it to the actual build order from the control group. We perform our experiment at three different time intervals, each corresponding roughly to a phase in the game, early, mid and late.

These are the parameters we use for each run:

- **Match-up**:
  The match-up to test.
  Values: PvP, PvT, PvZ, TvP, TvT, TvZ, ZvP, ZvT, ZvZ
- **Noise**:
  The noise, as a percentage of the potential observations.
  Values: 0%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%
- **Time**:
  The current in-game time.
  Values: 150, 300, 500
- **Steps**:
  The number of lookahead steps.
  Values: 0,1,2,3

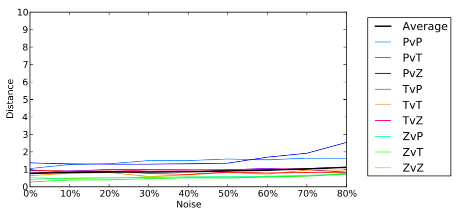
Each experiment outputs a number which is the average distance, see Section 6.3 for details, from the recognized build orders to the actual build orders. Analyzing these numbers will tell us how well the model is able to match observations to build orders, which phases of the game it performs best in, how well our model handles noise and how much lookahead we can perform without diverging too far from the actual build order.

6.1.2 Results

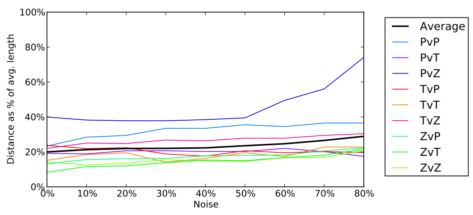
All the raw numbers from the experiments are listed in Section A.1. From these numbers we have highlighted the important trends as graphical representations. **NOTE:** All graphs are available in full size in Appendix B.

At $t = 150$, as illustrated in Figure 6.1, we see the distance go from 15% to 25% as the noise goes from 0% to 80% for most match-ups, which tells us that noise has a low impact on the distance. The spread of the different match-ups is fairly low, with most match-ups having a similar curve to that of the average. The three Zerg match-ups have better results than the Protoss and Terran match-ups, with the PvZ match-up showing a distinctive increase in distance as the noise increases, driving the average up.

At $t = 300$, as illustrated in Figure 6.2, we see the same trends as we did at $t = 150$.

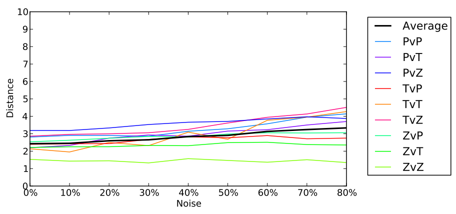


(a) with absolute distance.

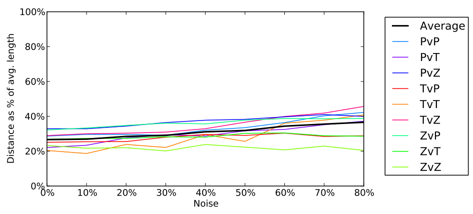


(b) with distance as % of average build order length per race.

Figure 6.1: Results of build order recognition at $t = 150$.



(a) with absolute distance.



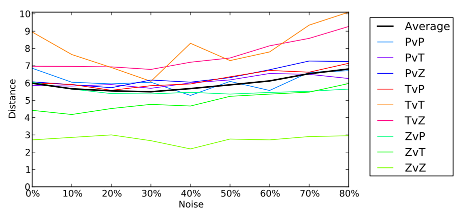
(b) with distance as % of average build order length per race.

Figure 6.2: Results of build order recognition at $t = 300$.

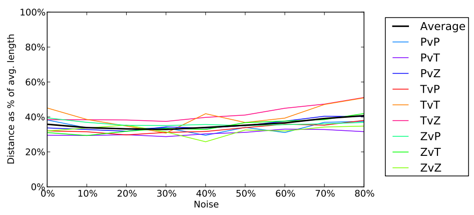
The spread of the match-ups has gone down and the variation of distance over the variation of noise is still low. Comparing the average distance at $t = 300$ to that in $t = 150$ we see a distinct increase across all noises. The Zerg match-ups continues to have better results compared to the Terran and Protoss match-ups, but the difference is less distinct.

At $t = 500$, as illustrated in Figure 6.3, the trend from $t = 150$ and $t = 300$ continues. The spread of the match-ups decreases and the distance over noise is increased for all noises. One interesting thing to note here, is that the average distance at 0% noise is higher than the average distance at 10% to 40% noise. Also, the Zerg match-ups no longer have a distinctly better result than the Protoss and Terran match-ups.

Comparing the data of the three time-steps we notice that noise has a generally low impact on the distance, on all match-ups. We do see that time, and by extension length of build orders as we can see in Table 6.1, however, has a high impact on distance. Also, as the length of build orders increase to a certain point the difference between the match-ups is almost gone. As we can see in Figure 6.4 Zerg has a distinctively shorter average length of build orders, which is caused by the difference in mechanics for producing units. This is causing the Zerg match-ups to perform better up to and including $t = 300$. From this we can also see that



(a) with absolute distance.



(b) with distance as % of average build order length per race.

Figure 6.3: Results of build order recognition at $t = 500$.

representing distance as an absolute number is prone to misrepresentation of the results, because of the high variance in build order lengths between match-ups and over time.

Table 6.1: Average length of player build orders per match up.

Player race	**Time**	**Opponent race**			**Average**
		P	**T**	**Z**	
P	**150**	4.480	4.805	3.428	4.238
	300	9.807	9.955	9.693	9.818
	500	17.921	19.832	18.007	18.587
T	**150**	4.057	4.195	3.592	3.948
	300	9.539	10.468	9.88	9.962
	500	18.78	19.832	18.129	18.914
Z	**150**	3.216	3.348	3.687	3.417
	300	7.889	8.244	6.587	7.573
	500	15.321	14.24	8.489	12.683

As we compare how lookahead and noise impacts the different match-ups, we notice the same recurring trend of noise having a low impact on the distance as in the recognition tests. This is most apparent in the non-mirror match-ups, illustrated in Figure 6.6. The mirror match-ups, illustrated in Figure 6.5, display a more erratic behavior at $t = 500$, but at $t = 150$ and $t = 300$ the graphs behave the same way as the non-mirror match-ups. The interesting thing to take from these results is that changes in time presents itself as clear jumps in distance for most match-ups, which further illustrates that time and length of build orders have more impact on distance than noise in observations. The distance is generally strictly increased by increasing the lookahead, as we can see on the graphs in Figure 6.5 and Figure 6.6 by looking at how there are few intersecting lines.

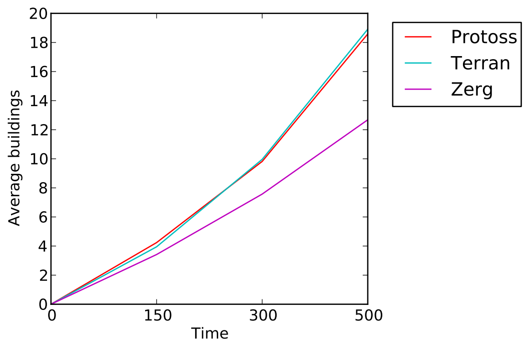


Figure 6.4: The average length of a build order per race over time.

6.2 Strategic Reasoning

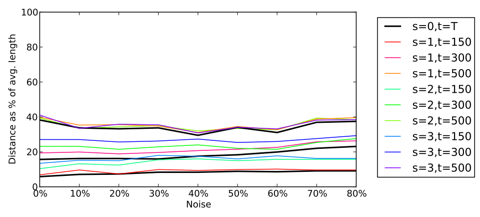
The main purpose of our strategic reasoning model is to select actions that are good responses to the strategy the opponent is following, and to see if the model is capable of this we need to test if the model takes good choices. The aim of this experiment is to gather key numbers that will tell us how our model does in regard to this purpose.

6.2.1 Setup

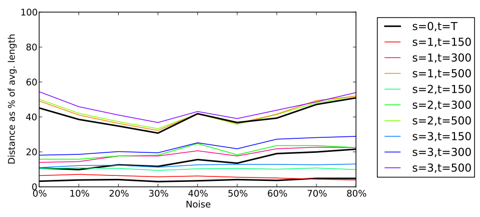
For this experiment we perform a cross-validation test for each match-up, with an opponent model learned from 90% of the available material and tested with the remaining 10%. For mirror match-ups we use each replay two times, one time for each player perspective, which means we only need 5% of the original amount to construct a sufficient test set. Build orders from the player gathered from the test set replays serve as the control group. To look up the proper counter plan we use the build order of the opponent gathered from the test set replays without any noise, this is because we are not interested in testing the recognizing capabilities of the opponent model for this experiment. Once a counter plan has been selected we compare it to the actual build order from the control group. We perform our experiment at three different time intervals, each corresponding roughly to a phase in the game, early, mid and late.

These are the parameters we use for each run:

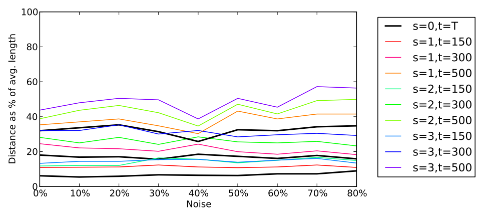
- **Match-up**:
  The match-up to test.
  Values: PvP, PvT, PvZ, TvP, TvT, TvZ, ZvP, ZvT, ZvZ
- **Time**:



(a) PvP



(b) TvT



(c) ZvZ

Figure 6.5: Comparison of noise impact on lookahead in mirror match-ups as percentage of average build order length over time.

The current in-game time.
Values: 150, 300, 500

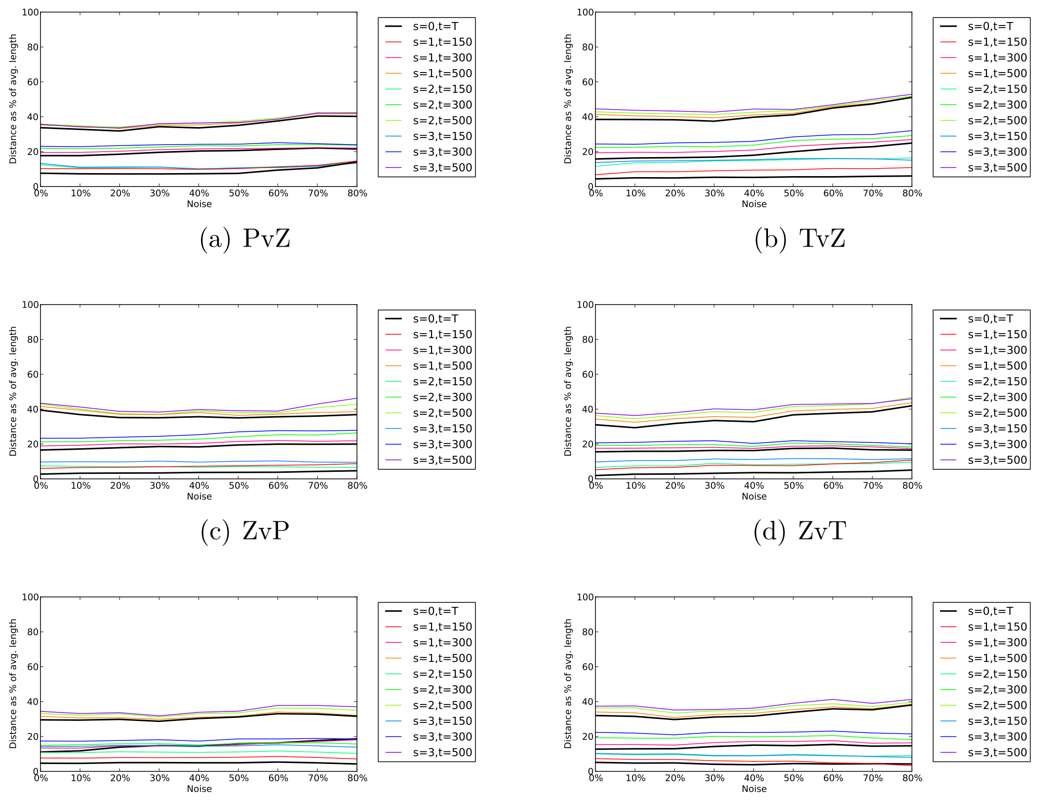
- **Steps**:
  The number of lookahead steps.
  Values: 0,1,2,3

Each experiment outputs a number which is the average distance, see Section 6.3 for details, from the chosen build orders to the actual build orders. Comparing these numbers tells us if the model is making good choices, since we base our theories on the assumptions that good players make good choices, and how much lookahead we can perform without diverging too much from the actual build order.

6.2.2 Results

All the raw numbers from the experiments are listed in Section A.2. From these numbers we have highlighted the important trends as graphical representations. **NOTE:** All graphs are available in full size in Appendix B.

At $t = 150$ we see that increasing the length of our selected counter-plan, analogous to the lookahead in plan recognition in the opponent model tests, increases the distance from the counter-plans used by the players in the test examples, illustrated in Figure 6.7(a). At $t = 300$ the increase in distance with the increase in length is less apparent, see Figure 6.7(b), and at $t = 500$ the increase is almost non-existent,



(a) PvZ (b) TvZ

(c) ZvP (d) ZvT

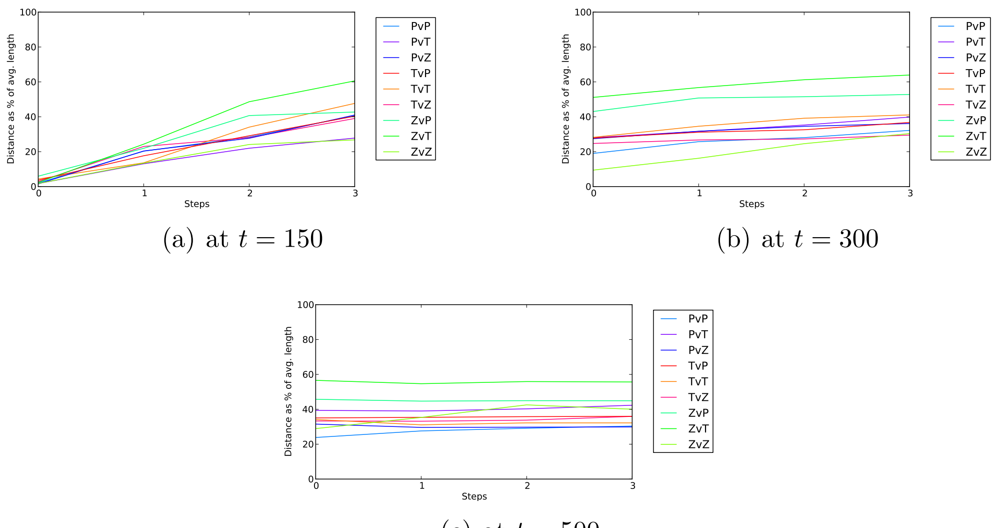
(e) PvT (f) TvP

Figure 6.6: Comparison of noise impact on lookahead in non-mirror match-ups as percentage of average build order length over time.

see Figure 6.7(c).

The most notable difference between the times in this experiment is the difference at $s = 0$. This is a measure of how well the build order of the player is contained within the model at the chosen point in time. A distance of 0 means that the current build order is fully contained within the tree, a distance of more than 0 means that some buildings in the tree are not in our build order. At $t = 150$ the distance at $s = 0$ is close to 0% for all match-ups. At $t = 300$ this distance has increased to between 20% and 25% for most match-ups and at $t = 500$ the distance has increased to 25% to 40%.

Interestingly, we see that the Zerg match-ups, which we had good results for in the opponent modeling experiments, are performing worse than the Terran and Protoss match-ups in this experiment. The increase in distance over time does not have the same distinctive jumps as we saw with the opponent model experiment, however we do see how much of an impact the length of the build order has. In Figure 6.7(a) we see how increasing length from an average of four, see Table 6.1,



(a) at $t = 150$ (b) at $t = 300$ (c) at $t = 500$

Figure 6.7: Results of counter-tree recognition with distance as % of average build order length per race.

to roughly seven by adding three additional steps to the plan increases the average distance to almost 40% from about 5%. After this point the distance does not vary much with the increasing length of build orders.

6.3 Distance Metric

To measure the accuracy of the opponent build order recognition and build order selection we use a set distance defined by Equation 6.1. This metric is the same as the one used by Synnaeve and Bessière[29], and it measures the distance between two sets of buildings. If there is one more or less building in the recognized build order than in the actual build order, it is a distance of 1. If the recognized build order has the same buildings as the actual build order, but one building is different, it is a distance of 2, etc.

$$\{BO_{recognized} \cup BO_{actual}\} \setminus \{BO_{recognized} \cap BO_{actual}\} \tag{6.1}$$

When recognizing a build order it is first important to know which buildings the opponent has, and second which order they were constructed in. By knowing which buildings the opponent has, you can infer which units he has available for production, and by knowing the amount of supply buildings, know how many units he has. The order says how much time he has had available to make use of the buildings. An early Protoss Cybernetic Core means that he has had more time to construct Protoss Dragoons than a late one. When observing a building it is not possible to know when it was constructed, only that it exists, which means that it is not possible to say which order the observed buildings were created other than inferring it through the observed times. This metric says how accurate our recognition of the opponents build order is, which is most important. Which buildings the opponent has is measured using the set distance, and which order they were constructed in is taken into account by using the timing of the states, and the timing of the observations. This way we account for both the important parts of inferring a build order.

An alternative to this metric is to see for each step in the build order if the correct building was recognized. The problem using this metric is that a single building can shift the entire build order and lead to a complete miss, even though the build order contains the same buildings in the exact same sequence, except for one building. Getting a good result using this metric means that the build order is more or less spot on, however, getting a bad result does not necessarily mean that the recognition was bad.

The problem with our metric is that it is not possible to know if the missed buildings are important game changing buildings or buildings with small impact on the game play. Being game changing can come from a building representing a clear shift in strategy, for instance moving from ground units to air units, or from the element of surprise by being undiscovered. Both types of game changing buildings could potentially be included in a distance metric by adding weights to certain buildings according to a set of rules, this would, however, require high knowledge of different strategies in Starcraft. This would make the metric much more complex and would make the results harder to interpret.

6.3.1 Distance Metric Examples

Recognized:
`{Protoss Gateway, Protoss Pylon, Protoss Pylon}`
Actual:
`{Protoss Gateway, Protoss Pylon, Protoss Pylon, Protoss Assimilator}`
A distance of $d = 1$, as the predicted build order does not contain the `Protoss Assimilator`.

Recognized:
`{Protoss Gateway, Protoss Pylon, Protoss Pylon, Protoss Cybernetics Core}`
Actual:
`{Protoss Gateway, Protoss Pylon, Protoss Pylon, Protoss Assimilator}`
A distance of $d = 2$, as the predicted build order does not contain the `Protoss`

`Assimilator`, and it contains the `Protoss Cybernetics Core` that the actual build order does not contain.

Recognized:
`{Protoss Gateway, Protoss Pylon, Protoss Pylon, Protoss Cybernetics Core, Protoss Forge}`
Actual:
`{Protoss Gateway, Protoss Pylon, Protoss Pylon, Protoss Assimilator}`
A distance of $d = 3$, as the predicted build order does not contain the `Protoss Assimilator`, and it contains the `Protoss Cybernetics Core` and `Protoss Forge` that the actual build order does not contain.

Chapter 7

Discussion

In this chapter we discuss the results of our experiments. Section 7.1 contains a discussion of the results of our opponent model. Section 7.2 contains a discussion of the results of our strategic reasoning.

7.1 Opponent Modeling

As we see from the experiments, time has a large impact on the distance from recognized build order to actual build order. Time in our experiments represents the different phases on the game, from early to mid to late. Another metric that also represents the phases in a game well is length of build orders. These two metrics are proportional to each other, which is why we only used time in the experiments. Looking at Table 6.1 and Figure 6.4 we see how the length of the build orders increase when time increases.

As the length of the build order increases, so does the potential difference between two build orders. In addition, when playing Starcraft, games tend to change towards a more reactive action selection rather than a planned action selection. This is due to the number of potential actions increasing as the game progresses which makes it harder to predict actions. This leads to less static build orders and more dynamic and reactive build orders, which is reflected in how the distance increases with time in our experiments.

The reason for including a noise parameter was to see how well our model performed with incomplete information. By looking at the graphs in Figure 6.1(b), Figure 6.2(b) and Figure 6.3(b) we see that the model generally performs worse under increasing noise, but not by much compared to the changes in performance with time. Looking at the average numbers for build order recognition in Table A.10:

For $t = 150$, $s = 0$:
0% noise: 20%
80% noise: 28.9%
The difference: 8.9%

For For $t = 500$, $s = 0$:
0% noise: 35.8%
80% noise: 40.8%
The difference: 5%

The difference in distance over varying noise is very little compared to the difference over time, which is 15.8% for 0% noise and 11.8% for 80% noise, between $t = 150$ and $t = 500$. The difference in distance is even decreasing as time goes up. This makes it more apparent that length of build orders and time have a more significant impact on the performance of our opponent model. Also, this is a further indicator of build orders becoming less consistent with the length of the game, caused by increased complexity in strategic decision making.

Because of how construction of buildings must adhere to the tech tree mechanics of advanced buildings having basic buildings as prerequisites, we are guaranteed that a build order includes these basic buildings after a certain point in time. This makes it so that we are guaranteed a certain hit percentage for all test examples regardless of what we recognize the build order as. This is further reinforced because we have included supply buildings in the build orders, and supply is something every player needs to have. Supply buildings are more of a necessary evil late game, where the cost is low compared to available resources, but at the early stages of the game the resources are scarce and a supply building represent a significant cost. The combination of these elements is the reason for the variance in noise dropping as the length of build orders increase.

Looking at how the Zerg match-ups tested in the experiment we see further evidence of how length of build orders impacts the results. Zerg has a very different mechanic for unit production, and therefore require less of the specialized buildings, making the length of the build order significantly lower than Terran and Protoss and resulting in overall better results in the experiment, see Figure 6.1 and Figure 6.2.

By looking at the graphs in Figure 6.5 and Figure 6.6 we can compare the impact of lookahead at the different times. Lookahead gives a significant increase in distance at $t = 150$, a less increase at $t = 300$ and almost no increase at $t = 500$.

For $t = 150$ at 40% noise:
$s = 0$: 23.5%
$s = 1$: 35.3%
$s = 2$: 48.5%
$s = 3$: 55.5%

For $t = 500$ at 40% noise:
$s = 0$: 34%
$s = 1$: 35.5%
$s = 2$: 37%
$s = 3$: 38%

By comparing the numbers at $t = 150$ to $t = 500$ we notice a sharp decline in increased distance between $s = 0$ and $s = 3$. At $t = 150$ the difference in distance is 32% whereas at $t = 500$ it is 4%. This further strengthens the theory that build order length is the most significant factor on distance in this experiment and that difference in build order length has a larger impact earlier in the game than later in the game.

From this we can extract a couple important points. First, we do see that most build orders share subsets of shorter build orders, which means that there are similarities between build orders that can be captured and used for recognition. In our experiment we used the fall-backs as described in Section 4.1.2, which coupled with our metric of set distance is polluting our results. In Figure 6.3(b) we see the average distance between 30% and 40%, pointing towards build orders having about 60% of their buildings in common at this point in time with our method of recognition. Second, we see that strict matching on build orders becomes less useful as the game progresses. Any advantage gained in the early stages of a game, in terms of build orders, dissipates as time increases. This is caused by how the tech tree mechanic makes advanced buildings depend on the presence of basic buildings. This points towards separate matching for late game advanced tech and early game basic tech openers being a good idea.

7.2 Strategic Reasoning

We see much of the same with the results of the strategic reasoning experiments as we did with the opponent modeling experiments. Time and build order length has high impacts on distance. Length has a higher impact earlier in the game than late in the game. A certain amount of the build order is guaranteed to match when build orders pass a certain length, because the same basic buildings appear in all build orders, leading to a high variance in distance at a earlier time in the game and a low variance in distance at a later time in the game.

Further reinforcing this is the results of the Zerg match-ups. The Zerg unit production mechanic means that there are less buildings making up a build order which again means that there are less common buildings in build orders as games progress. This is reflected by the Zerg match-ups having worse results than Terran and Protoss match-ups, see Figure 6.7.

Since this experiment was aimed at testing the sanity of the chosen plans, we are not saying anything about how well the plans would work in a live setting. Given a correct recognition of the opponent's build order, we know that the chosen build order has been used against that build order by someone, and therefore a miss on the correct build order is not a big problem, as there are several ways to play against a build order. A miss only indicates that we have chosen a different path than the current player, without necessarily saying that we chose a bad build order.

Our strategic reasoning model is built around the assumption that we can learn counter plans from replays and select counter plans based on the output from the opponent model. What we extracted from the opponent model experiments is that a change in the way build orders are represented in the model is needed, and this will have to be reflected in how build orders are represented in the strategic reasoning model. Selecting a plan for an opener is more dependent on the opponent's opener and selecting a plan for the late game is more dependent on the opponent's actions and behavior in the late game.

Chapter 8

Conclusion and Further Work

This chapter contains the concluding remarks to our thesis. Section 8.1 presents the result of our work in relation to our goals and the important discoveries made during the experiments. Section 8.2 contains the ideas we have for future improvements and solutions to the problems we discovered.

8.1 Conclusions

The goal of this thesis was to construct a model capable of performing opponent modeling and a model doing strategic reasoning by selecting plans that counter the opponent. We presented an approach to this problem that would require very little prior knowledge of the Starcraft domain. By learning build orders, the central defining component in Starcraft strategies, from demonstrations and structuring them in a model that is easily extended by adding demonstrations, we are able to perform much of the reasoning needed to play Starcraft at a basic level without having to resort to advanced techniques and models.

From our experiments we were able to determine that there are in fact repeating patterns in build orders that can be used to perform recognition of build orders. We were also able to determine that the distance between a recognized build order and the actual build order is primarily dependent on the length of the build order and the phase of the game. This points towards a change in mindset of the players, from a more deliberate and planned strategy early in the game to a more reactive and dynamic strategy later in the game.

Our experiments were not able to determine the importance of observations and time of observations. This is due to the metric of set distance used in the experiments not capturing the importance of order in build orders and the fall backs used in the model coupled with build orders sharing a significant subset of buildings causing pollution of the results.

8.2 Further Work

By including inferred buildings, instead of just the directly observed buildings, the accuracy of our predictions can be improved. An inferred building is a building that is not directly observed, but the effects of constructing the building are, such as a specific unit being available for the opponent, or the unit having an upgrade. This means that the relations between units, buildings, upgrades etc. have to be specified, but this is possible to do with the tools we have used.

One source of error in our model is the presence of supply buildings. As we have removed static defenses from the recognized buildings in a build order, only supply buildings are left as buildings that do not impact the technology path. Supply buildings are present for the reason that they act as markers between a tech rush and a slow and steady push. One way to remove supply buildings from the opponent model and still keep this distinction is by unifying buildings based on both previous buildings, and time as well. Our model assumes that all buildings that have the same previous buildings belong to the same tech path, regardless of the time they were built. An example of another way of doing it is the build orders 1:{`Terran Barracks(51), Terran Refinery(90)`}, 2:{`Terran Barracks(52), Terran Refinery(70)`} and 3:{`Terran Barracks(49), Terran Refinery(76)`}. Here we can see that the three different Terran Barracks can be counted as the same part of a build order, due to their closeness in time. However, the three Terran Refinery diverge quite a lot in their timing. Here a better unification of build order could be that build order 2 and 3 were counted as the same build order, while build order 1 was counted as a different one. Deciding which timings belong together can be done using a clustering algorithm[19]. The data that is available is one-dimensional, the only difference between each building is the timing.

One problem with evaluating our models is our distance metric. It does not take into account the ordering of buildings which our models are designed to capture. A distance metric that accounts for both the ordering as well as the contents of a build order will make it easier to evaluate and compare our model and other models for build order recognition and prediction.

What will enhance our model most is the addition of even more replays. The more replays that are used in building our models, the better they get. Scraping replays take time, the game is not as popular as it used to be and new replays are not uploaded that often.

As it stands now, our models do not warrant the work needed to implement them in a working bot. If our problems can be fixed, an implementation of them in a bot is warranted to see if they work in practice.

References

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

[2] D. Aha, M. Molineaux, and M. Ponsen. Learning to win: Case-based plan selection in a real-time strategy game. *Case-Based Reasoning Research and Development*, pages 5–20, 2005.

[3] D.W. Albrecht, I. Zukerman, and A.E. Nicholson. Bayesian models for keyhole plan recognition in an adventure game. *User modeling and user-adapted interaction*, 8(1):5–47, 1998.

[4] R. Arrabales, A. Ledezma, and A. Sanchis. CERA-CRANIUM: A test bed for machine consciousness research. 2009.

[5] R. Arrabales, A. Ledezma, and A. Sanchis. Towards conscious-like behavior in computer game characters. In *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, pages 217–224. IEEE, 2009.

[6] R.K. Balla and A. Fern. UCT for tactical assault planning in real-time strategy games. In *21st international joint conference on artificial intelligence*, pages 40–45, 2009.

[7] Alex Bellos. Rise of the e-sports superstars [online], 2011 (accessed Nov 2nd). `http://news.bbc.co.uk/2/hi/programmes/click_online/6252524.stm`.

[8] M. Buro and T. Furtak. RTS games and real-time AI research. In *Proceedings of the Behavior Representation in Modeling and Simulation Conference (BRIMS)*, volume 6370, 2004.

[9] E. Charniak and R.P. Goldman. A Bayesian model of plan recognition. *Artificial Intelligence*, 64(1):53–79, 1993.

[10] M. Chung, M. Buro, and J. Schaeffer. *Monte Carlo planning in RTS games.* PhD thesis, University of Alberta, 2005.

[11] D. Churchill and M. Buro. Build Order Optimization in StarCraft. In *Seventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2011.

[12] Dave Churchill. 2011 AIIDE StarCraft Competition - Cumulative Results [online], 2011 (accessed Nov 24th). `https://docs.google.com/spreadsheet/ccc?key=0An3xUNEy2rixdHUza1BzZE50cGdUdHEwbGlhSU5rckE#gid=0`.

[13] M. Fagan and P. Cunningham. Case-based plan recognition in computer games. *Case-Based Reasoning Research and Development*, pages 1066–1066, 2003.

[14] H. Hoang, S. Lee-Urban, and H. Muñoz-Avila. Hierarchical Plan Representations for Encoding Strategic Game AI. In *Proc. Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE-05)*, 2005.

[15] J.L. Hsieh and C.T. Sun. Building a player strategy model by analyzing replays of real-time strategy games. In *Neural Networks, 2008. IJCNN 2008.(IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*, pages 3106–3111. IEEE, 2008.

[16] ieee cig.org. CIG 2011 StarCraft RTS AI Competition [online], 2011 (accessed Nov 23rd). `http://ls11-www.cs.uni-dortmund.de/rts-competition/starcraft-cig2011`.

[17] F. Kabanza, P. Bellefeuille, F. Bisson, A.R. Benaskeur, and H. Irandoust. Opponent behaviour recognition for real-time strategy games. In *AAAI Workshops*, 2010.

[18] M. Mateas and A. Stern. A behavior language for story-based believable agents. *Intelligent Systems, IEEE*, 17(4):39–47, 2002.

[19] Matteo Matteucci. A Tutorial on Clustering Algorithms [online], 2012 (accessed June 3rd). `http://home.dei.polimi.it/matteucc/Clustering/tutorial_html/`.

[20] K. Mishra, S. Ontañón, and A. Ram. Situation assessment for plan retrieval in real-time strategy games. *Advances in Case-Based Reasoning*, pages 355–369, 2008.

[21] M. Molineaux, D.W. Aha, and P. Moore. Learning continuous action models in a real-time strategy environment. In *Proceedings of the Twenty-First Annual Conference of the Florida Artificial Intelligence Research Society*, pages 257–262, 2008.

[22] S. Ontanón, K. Bonnette, P. Mahindrakar, M.A. Gómez-Martín, K. Long, J. Radhakrishnan, R. Shah, and A. Ram. Learning from human demonstrations for real-time case-based planning. 2009.

[23] S. Ontañón, K. Mishra, N. Sugandh, and A. Ram. Case-based planning and execution for real-time strategy games. *Case-Based Reasoning Research and Development*, pages 164–178, 2007.

[24] M. Ponsen. *Improving adaptive game AI with evolutionary learning*. PhD thesis, Citeseer, 2004.

[25] M. Ponsen, H. Munoz-Avila, P. Spronck, and D.W. Aha. Automatically generating game tactics through evolutionary learning. *AI Magazine*, 27(3):75, 2006.

[26] M. Ramırez and H. Geffner. Plan recognition as planning. In *Proc. IJCAI*, pages 1778–1783, 2009.

[27] F. Schadd, S. Bakkes, and P. Spronck. Opponent modeling in real-time strategy games. In *8th International Conference on Intelligent Games and Simulation (GAME-ON 2007)*, pages 61–68, 2007.

[28] G. Synnaeve and P. Bessiere. A Bayesian model for opening prediction in RTS games with application to StarCraft. In *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, pages 281–288. IEEE, 2011.

[29] G. Synnaeve and P. Bessiere. A Bayesian Model for Plan Recognition in RTS Games Applied to StarCraft. In *Seventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2011.

[30] G. Synnaeve and P. Bessiere. A Bayesian model for RTS units control applied to StarCraft. In *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, pages 190–196. IEEE, 2011.

[31] H.J. van den Herik, H. Donkers, and P.H.M. Spronck. Opponent modelling and commercial games. In *IEEE 2005 Symposium on Computational Intelligence and Games*, pages 15–25. IEEE Press, Piscataway, NJ, USA, 2005.

[32] B. Weber and M. Mateas. Conceptual Neighborhoods for Retrieval in Case-Based Reasoning. *Case-Based Reasoning Research and Development*, pages 343–357, 2009.

[33] Ben G. Weber, Peter Mawhorter, Michael Mateas, and Arnav Jhala. Reactive planning idioms for multi-scale game AI. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, pages 115–122. IEEE, August 2010.

[34] B.G. Weber and M. Mateas. A data mining approach to strategy prediction. In *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, pages 140–147. IEEE, 2009.

[35] B.G. Weber and M. Mateas. Case-based reasoning for build order in real-time strategy games. In *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment Conference, AAAI Press*, pages 106–111, 2009.

[36] B.G. Weber, M. Mateas, and A. Jhala. Applying Goal-Driven Autonomy to StarCraft. In *Proceedings of the Sixth Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2010.

[37] B.G. Weber, M. Mateas, and A. Jhala. Case-based goal formulation. In *Proceedings of the AAAI Workshop on Goal-Driven Autonomy*, 2010.

[38] B.G. Weber, M. Mateas, and A. Jhala. Learning from Demonstration for Goal-Driven Autonomy. 2012.

[39] B.G. Weber and S. Ontanón. Using Automated Replay Annotation for Case-Based Planning in Games. In *Proceedings of the ICCBR Workshop on Computer Games*, 2010.

[40] Wikipedia.org. List of best-selling video games [online], 2011 (accessed Nov 2nd). `http://en.wikipedia.org/wiki/List_of_best-selling_video_games#PC`.

[41] S. Wintermute, J. Xu, and J.E. Laird. SORTS: A human-level approach to Real-Time Strategy AI. *Ann Arbor*, 1001:48109–2121, 2007.

Appendix A

Results

A.1 Opponent Modeling

These tables are the unprocessed results of the experiments conducted with the opponent model.

Table A.1: Results from PvP tests

Noise	**Steps**	**Time**		
		150	**300**	**500**
0%	0	1.045	2.810	6.857
	1	1.227	3.476	7.143
	2	1.864	4.150	7.053
	3	2.409	4.850	7.368
10%	0	1.273	2.905	6.048
	1	1.727	3.571	6.333
	2	2.364	4.150	6.000
	3	2.727	4.850	6.000
20%	0	1.318	2.905	5.952
	1	1.318	3.381	6.381
	2	2.227	3.850	6.158
	3	2.682	4.600	6.421
30%	0	1.500	2.857	6.048
	1	1.773	3.524	6.238
	2	2.773	4.100	6.263
	3	3.227	4.700	6.368
40%	0	1.500	3.143	5.286
	1	1.682	3.714	5.571
	2	2.864	4.300	5.737
	3	3.136	4.900	5.526
50%	0	1.591	3.286	6.095
	1	1.773	3.857	6.190
	2	2.682	3.950	6.000
	3	2.864	4.550	6.105
60%	0	1.545	3.571	5.571
	1	1.818	4.048	5.857
	2	2.818	3.850	5.842
	3	3.182	4.650	5.947
70%	0	1.636	3.952	6.619
	1	1.727	4.619	6.952
	2	2.818	4.550	7.053
	3	2.909	4.950	6.842
80%	0	1.636	4.143	6.714
	1	1.727	4.714	7.095
	2	2.818	4.950	6.895
	3	2.909	5.250	6.895

Table A.2: Results from PvT tests

Noise	**Steps**	**Time**		
		150	**300**	**500**
0%	0	0.921	2.200	5.856
	1	1.514	2.656	6.260
	2	2.144	2.958	6.578
	3	2.833	3.445	6.822
10%	0	0.907	2.330	5.827
	1	1.500	2.633	6.077
	2	2.130	3.023	6.374
	3	2.837	3.427	6.574
20%	0	0.991	2.749	5.913
	1	1.565	2.898	6.101
	2	2.233	3.136	6.544
	3	2.949	3.507	6.673
30%	0	0.991	2.930	5.697
	1	1.556	2.874	5.918
	2	2.186	3.150	6.184
	3	2.930	3.597	6.312
40%	0	0.972	2.860	6.010
	1	1.565	2.870	6.154
	2	2.158	3.000	6.544
	3	2.856	3.441	6.718
50%	0	0.972	3.153	6.183
	1	1.602	3.023	6.264
	2	2.205	3.164	6.646
	3	2.893	3.687	6.842
60%	0	1.056	3.233	6.553
	1	1.694	3.251	6.745
	2	2.316	3.243	7.199
	3	3.023	3.687	7.500
70%	0	0.968	3.502	6.505
	1	1.579	3.400	6.644
	2	2.191	3.252	7.175
	3	2.898	3.744	7.490
80%	0	0.838	3.702	6.264
	1	1.412	3.581	6.370
	2	2.042	3.112	6.942
	3	2.749	3.682	7.342

Table A.3: Results from PvZ tests

Noise	**Steps**	**Time**		
		150	**300**	**500**
0%	0	1.370	3.188	6.072
	1	1.841	3.513	6.354
	2	2.235	3.946	6.452
	3	2.390	4.150	6.417
10%	0	1.310	3.186	5.907
	1	1.830	3.522	6.206
	2	1.924	3.896	6.267
	3	1.986	4.107	6.154
20%	0	1.296	3.333	5.732
	1	1.849	3.641	5.993
	2	1.952	3.955	6.128
	3	2.023	4.226	6.049
30%	0	1.296	3.530	6.175
	1	1.838	3.812	6.316
	2	2.003	4.074	6.438
	3	2.014	4.309	6.485
40%	0	1.318	3.661	6.052
	1	1.767	3.881	6.361
	2	1.849	4.214	6.534
	3	1.792	4.358	6.560
50%	0	1.353	3.710	6.316
	1	1.841	3.916	6.546
	2	1.905	4.182	6.730
	3	1.932	4.370	6.613
60%	0	1.696	3.849	6.770
	1	2.014	3.945	6.955
	2	2.062	4.310	7.075
	3	1.952	4.526	6.951
70%	0	1.921	3.974	7.275
	1	2.184	4.014	7.515
	2	2.162	4.330	7.580
	3	2.094	4.422	7.605
80%	0	2.540	3.875	7.244
	1	2.638	3.968	7.498
	2	2.507	4.259	7.651
	3	2.402	4.309	7.586

Table A.4: Results from TvP tests

Noise	**Steps**	**Time**		
		150	**300**	**500**
0%	0	0.963	2.392	6.010
	1	1.372	2.892	6.390
	2	1.884	3.627	6.816
	3	1.925	4.202	7.020
10%	0	0.888	2.425	5.919
	1	1.279	2.882	6.310
	2	1.781	3.575	6.831
	3	1.850	4.115	7.040
20%	0	0.907	2.434	5.581
	1	1.288	2.830	5.800
	2	1.809	3.566	6.280
	3	1.888	3.933	6.604
30%	0	0.763	2.675	5.848
	1	1.144	3.094	6.133
	2	1.647	3.764	6.498
	3	1.706	4.197	6.653
40%	0	0.712	2.825	5.948
	1	1.074	3.222	6.243
	2	1.660	3.717	6.585
	3	1.692	4.183	6.807
50%	0	0.837	2.769	6.362
	1	1.098	3.222	6.657
	2	1.740	3.755	7.068
	3	1.804	4.226	7.332
60%	0	0.791	2.896	6.729
	1	0.912	3.302	6.929
	2	1.665	3.858	7.275
	3	1.710	4.337	7.748
70%	0	0.819	2.712	6.629
	1	0.847	3.024	6.748
	2	1.628	3.599	6.981
	3	1.589	4.130	7.322
80%	0	0.795	2.750	7.143
	1	0.609	3.071	7.219
	2	1.698	3.410	7.444
	3	1.509	4.029	7.743

Table A.5: Results from TvT tests

Noise	**Steps**	**Time**		
		150	**300**	**500**
0%	0	0.636	2.136	8.950
	1	1.273	2.773	9.750
	2	2.091	3.136	9.950
	3	2.182	3.591	10.800
10%	0	0.773	1.955	7.650
	1	1.409	2.864	8.150
	2	2.136	3.136	8.350
	3	2.409	3.682	9.100
20%	0	0.818	2.500	6.900
	1	1.273	3.500	7.200
	2	2.091	3.500	7.400
	3	2.455	4.000	8.150
30%	0	0.591	2.318	6.100
	1	1.136	3.500	6.400
	2	1.864	3.591	6.600
	3	2.227	3.864	7.300
40%	0	0.682	3.091	8.300
	1	1.227	4.091	8.300
	2	2.045	4.909	8.300
	3	2.500	5.000	8.550
50%	0	0.818	2.682	7.300
	1	1.091	3.500	7.200
	2	2.091	3.636	7.100
	3	2.545	4.318	7.750
60%	0	0.727	3.773	7.800
	1	1.000	4.318	8.250
	2	2.000	4.682	8.200
	3	2.545	5.409	8.700
70%	0	0.955	3.955	9.350
	1	0.864	4.500	9.800
	2	2.136	4.682	9.550
	3	2.500	5.591	9.650
80%	0	0.955	4.273	10.100
	1	0.773	4.455	10.300
	2	1.955	4.455	10.300
	3	2.591	5.727	10.700

Table A.6: Results from TvZ tests

Noise	**Steps**	**Time**		
		150	**300**	**500**
0%	0	0.789	2.859	6.973
	1	1.228	3.501	7.487
	2	2.098	4.074	7.762
	3	2.482	4.421	8.076
10%	0	0.900	2.965	6.963
	1	1.534	3.559	7.349
	2	2.470	4.077	7.633
	3	2.655	4.388	7.924
20%	0	0.889	2.997	6.936
	1	1.528	3.539	7.275
	2	2.503	4.175	7.556
	3	2.672	4.536	7.841
30%	0	0.962	3.058	6.789
	1	1.634	3.643	7.141
	2	2.658	4.122	7.472
	3	2.725	4.582	7.733
40%	0	0.946	3.251	7.205
	1	1.705	3.793	7.443
	2	2.702	4.303	7.720
	3	2.796	4.694	8.054
50%	0	1.000	3.617	7.450
	1	1.732	4.184	7.614
	2	2.801	4.774	7.899
	3	2.894	5.155	8.011
60%	0	1.000	3.945	8.154
	1	1.878	4.403	8.302
	2	2.885	4.923	8.357
	3	2.913	5.373	8.498
70%	0	1.057	4.135	8.584
	1	1.854	4.602	8.654
	2	2.863	4.970	8.899
	3	2.880	5.406	9.065
80%	0	1.092	4.513	9.272
	1	1.970	4.882	9.225
	2	2.978	5.294	9.381
	3	2.734	5.806	9.588

Table A.7: Results from ZvP tests

Noise	**Steps**	**Time**		
		150	**300**	**500**
0%	0	0.423	2.538	6.046
	1	0.922	2.891	6.370
	2	1.153	3.256	6.590
	3	1.487	3.576	6.652
10%	0	0.501	2.628	5.655
	1	1.000	2.961	6.035
	2	1.101	3.262	6.142
	3	1.513	3.579	6.320
20%	0	0.515	2.743	5.391
	1	1.020	3.079	5.680
	2	1.072	3.360	5.728
	3	1.490	3.666	5.934
30%	0	0.524	2.846	5.373
	1	1.062	3.060	5.648
	2	1.084	3.401	5.672
	3	1.567	3.743	5.875
40%	0	0.569	2.813	5.465
	1	1.118	3.121	5.835
	2	1.014	3.495	5.970
	3	1.475	3.884	6.098
50%	0	0.577	2.982	5.363
	1	1.154	3.293	5.585
	2	1.095	3.703	5.813
	3	1.552	4.135	5.988
60%	0	0.588	3.057	5.458
	1	1.204	3.375	5.690
	2	1.061	3.880	5.810
	3	1.582	4.244	5.961
70%	0	0.661	3.045	5.525
	1	1.238	3.308	5.831
	2	1.029	3.864	6.269
	3	1.475	4.219	6.582
80%	0	0.717	3.060	5.651
	1	1.339	3.350	5.901
	2	1.012	4.044	6.560
	3	1.463	4.267	7.090

Table A.8: Results from ZvT tests

Noise	**Steps**	**Time**		
		150	**300**	**500**
0%	0	0.278	2.212	4.422
	1	0.752	2.503	4.889
	2	0.942	2.730	5.174
	3	1.385	2.949	5.369
10%	0	0.388	2.253	4.182
	1	0.911	2.512	4.625
	2	1.064	2.739	4.907
	3	1.501	2.984	5.177
20%	0	0.402	2.256	4.527
	1	0.941	2.541	4.916
	2	1.056	2.791	5.199
	3	1.499	3.074	5.408
30%	0	0.461	2.329	4.767
	1	1.119	2.574	5.091
	2	1.269	2.801	5.495
	3	1.629	3.119	5.719
40%	0	0.512	2.315	4.672
	1	1.094	2.491	5.030
	2	1.158	2.660	5.416
	3	1.586	2.900	5.646
50%	0	0.501	2.491	5.233
	1	1.078	2.674	5.554
	2	1.200	2.923	5.915
	3	1.663	3.119	6.077
60%	0	0.558	2.512	5.361
	1	1.226	2.700	5.679
	2	1.228	2.871	5.982
	3	1.649	3.045	6.115
70%	0	0.606	2.379	5.476
	1	1.329	2.615	5.753
	2	1.247	2.736	6.132
	3	1.578	2.977	6.162
80%	0	0.722	2.356	5.973
	1	1.536	2.474	6.226
	2	1.358	2.604	6.665
	3	1.646	2.865	6.550

Table A.9: Results from ZvZ tests

Noise	**Steps**	**Time**		
		150	**300**	**500**
0%	0	0.524	1.531	2.714
	1	0.929	2.082	3.000
	2	1.000	2.390	3.294
	3	1.127	2.724	3.714
10%	0	0.464	1.429	2.857
	1	0.940	1.878	3.143
	2	1.025	2.122	3.706
	3	1.225	2.724	4.071
20%	0	0.500	1.449	3.000
	1	0.952	1.837	3.286
	2	1.025	2.390	3.941
	3	1.225	3.000	4.286
30%	0	0.571	1.327	2.667
	1	1.048	1.714	2.952
	2	1.392	2.049	3.588
	3	1.310	2.552	4.214
40%	0	0.548	1.571	2.190
	1	0.952	2.061	2.571
	2	1.329	2.415	2.941
	3	1.324	2.724	3.286
50%	0	0.536	1.469	2.762
	1	0.917	1.694	3.667
	2	1.152	2.171	4.000
	3	1.183	2.414	4.286
60%	0	0.619	1.367	2.714
	1	0.952	1.571	3.286
	2	1.278	2.122	3.529
	3	1.282	2.517	3.857
70%	0	0.619	1.510	2.905
	1	1.048	1.735	3.524
	2	1.418	2.195	4.176
	3	1.366	2.586	4.857
80%	0	0.762	1.347	2.952
	1	0.929	1.551	3.524
	2	1.266	1.976	4.235
	3	1.141	2.483	4.786

Table A.10: The average numbers of all the match-ups.

Noise	**Steps**	**Time**		
		150	**300**	**500**
0%	0	0.772	2.430	5.989
	1	1.229	2.921	6.405
	2	1.712	3.363	6.630
	3	2.024	3.768	6.915
10%	0	0.823	2.453	5.668
	1	1.348	2.931	6.025
	2	1.777	3.331	6.246
	3	2.078	3.762	6.484
20%	0	0.848	2.596	5.548
	1	1.304	3.027	5.848
	2	1.774	3.414	6.104
	3	2.098	3.838	6.374
30%	0	0.851	2.652	5.496
	1	1.368	3.088	5.760
	2	1.875	3.450	6.023
	3	2.148	3.851	6.295
40%	0	0.862	2.837	5.681
	1	1.354	3.249	5.945
	2	1.864	3.668	6.194
	3	2.129	4.009	6.361
50%	0	0.909	2.907	5.896
	1	1.365	3.263	6.142
	2	1.875	3.584	6.352
	3	2.148	3.997	6.556
60%	0	0.953	3.134	6.123
	1	1.411	3.435	6.410
	2	1.924	3.749	6.585
	3	2.204	4.199	6.809
70%	0	1.027	3.240	6.541
	1	1.408	3.535	6.825
	2	1.944	3.798	7.091
	3	2.143	4.225	7.286
80%	0	1.117	3.335	6.813
	1	1.437	3.561	7.040
	2	1.959	3.789	7.341
	3	2.127	4.269	7.587

A.2 Strategic Reasoning

These tables are the unprocessed result of the experiments conducted with the strategic reasoning model.

Table A.11: Results of counter tree recognition.

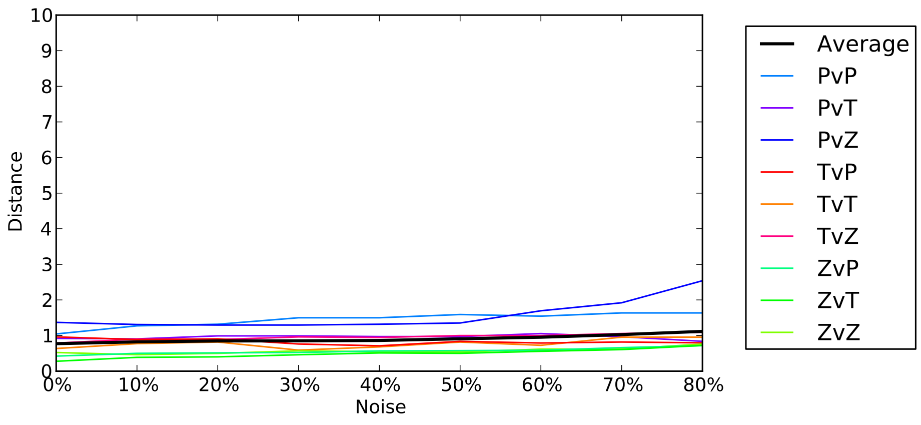
Match up	**Steps**	**Time**		
		150	**300**	**500**
PvP	0	0.091	1.864	4.273
	1	0.909	2.524	4.947
	2	1.273	2.750	5.222
	3	1.818	3.158	5.444
PvT	0	0.083	2.792	7.810
	1	0.628	3.149	7.738
	2	1.056	3.510	7.985
	3	1.335	3.966	8.394
PvZ	0	0.052	2.659	5.675
	1	0.699	3.074	5.333
	2	0.949	3.346	5.358
	3	1.406	3.502	5.382
TvP	0	0.148	2.651	6.583
	1	0.716	2.972	6.646
	2	1.181	3.105	6.722
	3	1.636	3.507	6.756
TvT	0	0.182	2.955	6.773
	1	0.571	3.619	6.167
	2	1.429	4.100	6.389
	3	2.000	4.300	6.389
TvZ	0	0.102	2.441	6.035
	1	0.826	2.640	6.014
	2	1.008	2.696	6.126
	3	1.402	2.914	6.514
ZvP	0	0.191	3.395	7.008
	1	0.705	4.006	6.847
	2	1.309	4.062	6.887
	3	1.374	4.165	6.880
ZvT	0	0.080	4.214	8.060
	1	0.814	4.678	7.786
	2	1.628	5.048	7.960
	3	2.028	5.270	7.927
ZvZ	0	0.065	0.620	2.457
	1	0.494	1.069	3.000
	2	0.889	1.619	3.611
	3	0.987	2.000	3.400

Appendix B

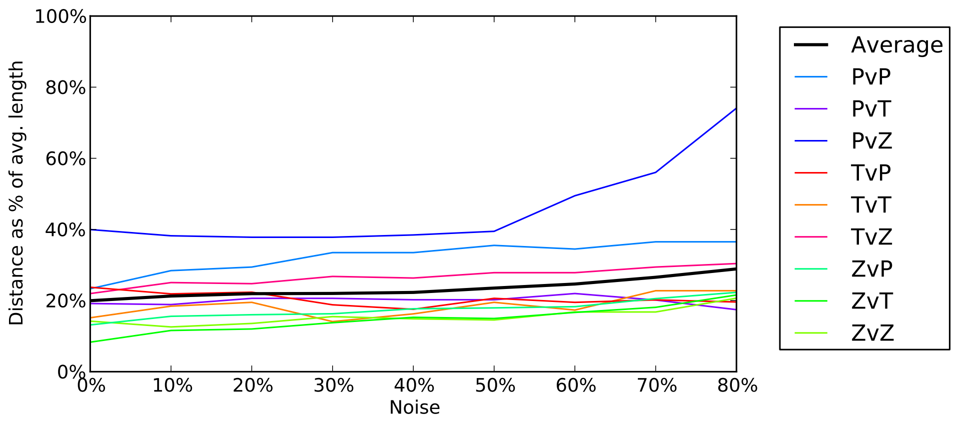
Graphs

B.1 Opponent Modeling

The graphs created from the results of the experiments on the opponent model.

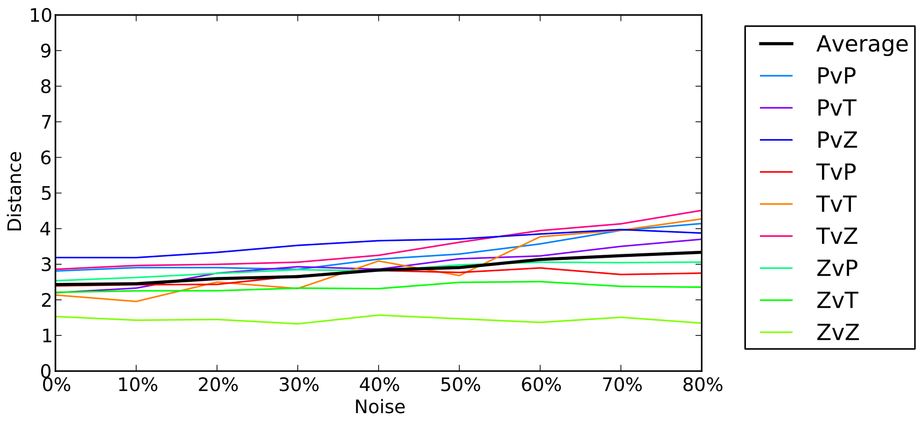


(a) with absolute distance.

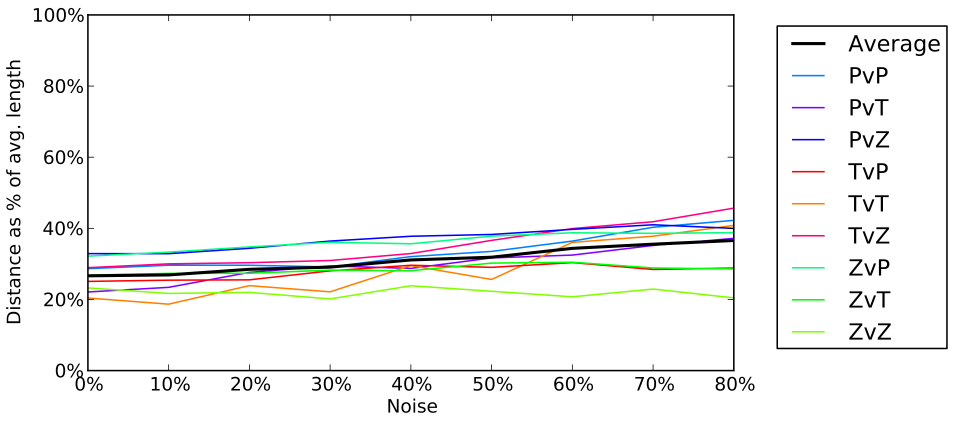


(b) with distance as % of average build order length per race.

Figure B.1: Results of build order recognition at $t = 150$.

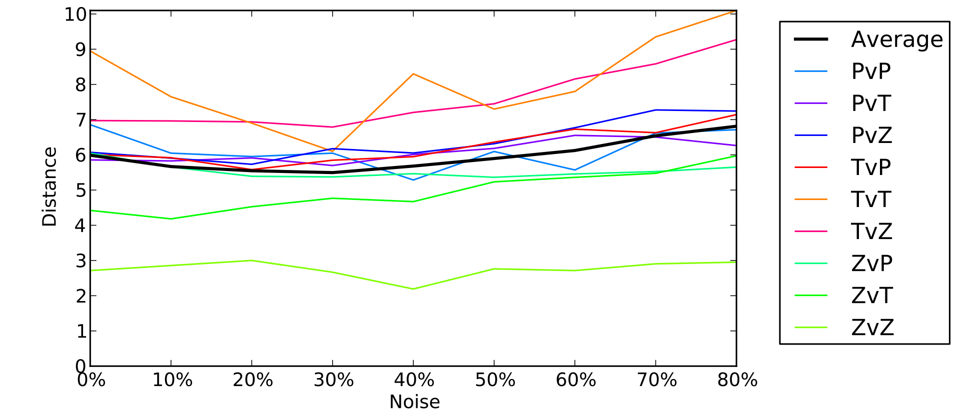


(a) with absolute distance.

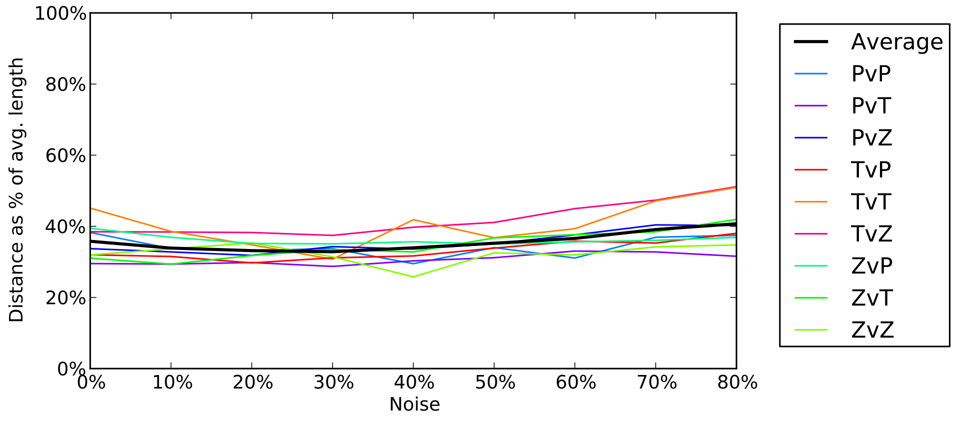


(b) with distance as % of average build order length per race.

Figure B.2: Results of build order recognition at $t = 300$.

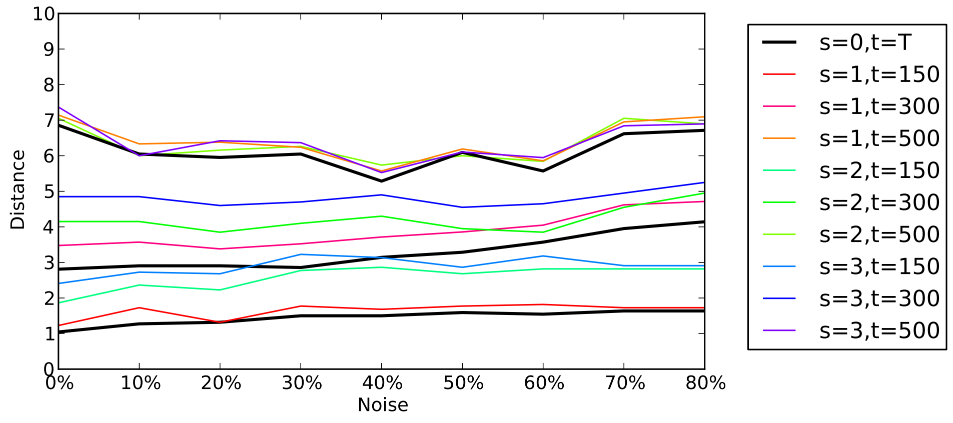


(a) with absolute distance.

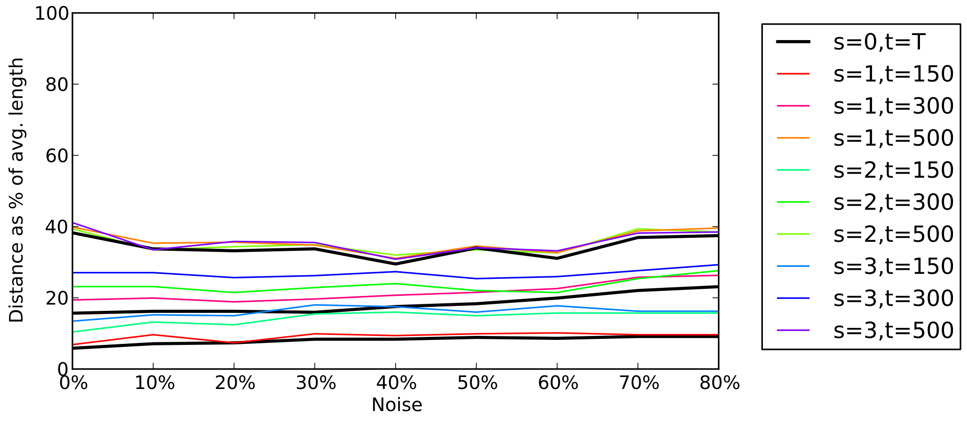


(b) with distance as % of average build order length per race.

Figure B.3: Results of build order recognition at $t = 500$.

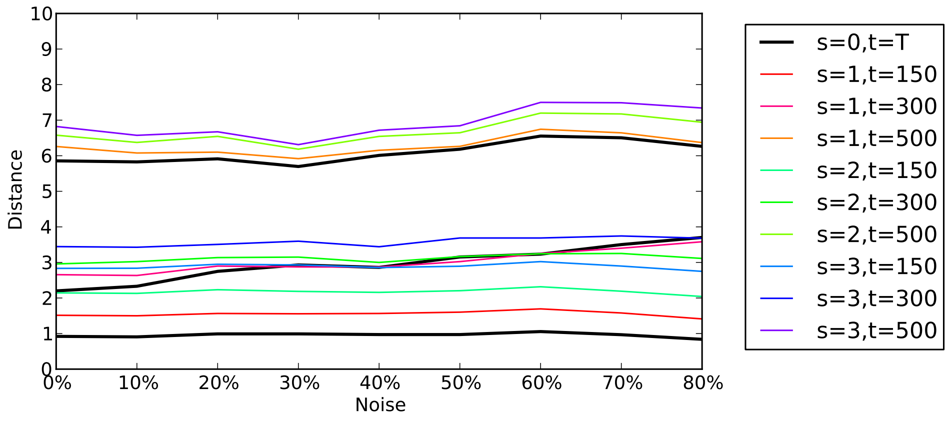


(a) with absolute distance.

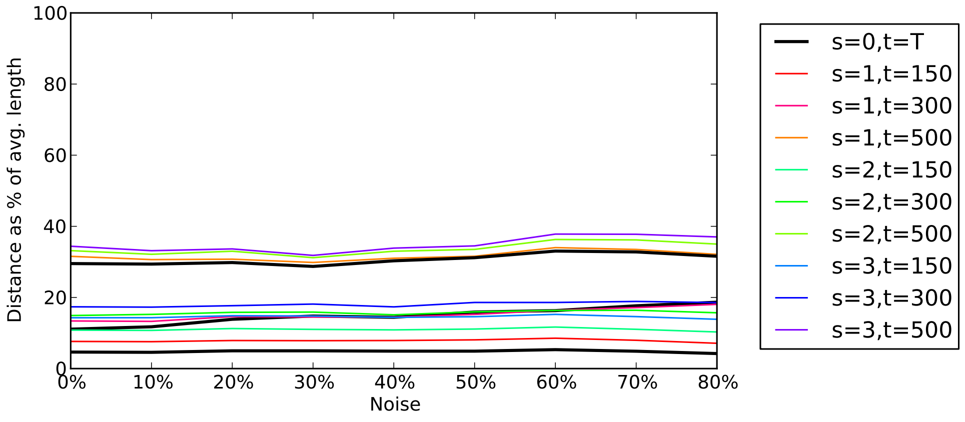


(b) with distance as % of average build order length per race.

Figure B.4: Impact of noise on predictions in Protoss vs Protoss.

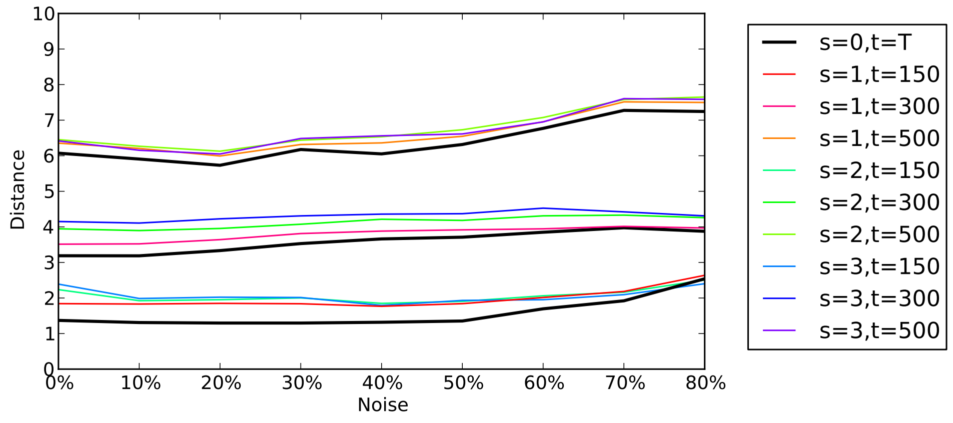


(a) with absolute distance.

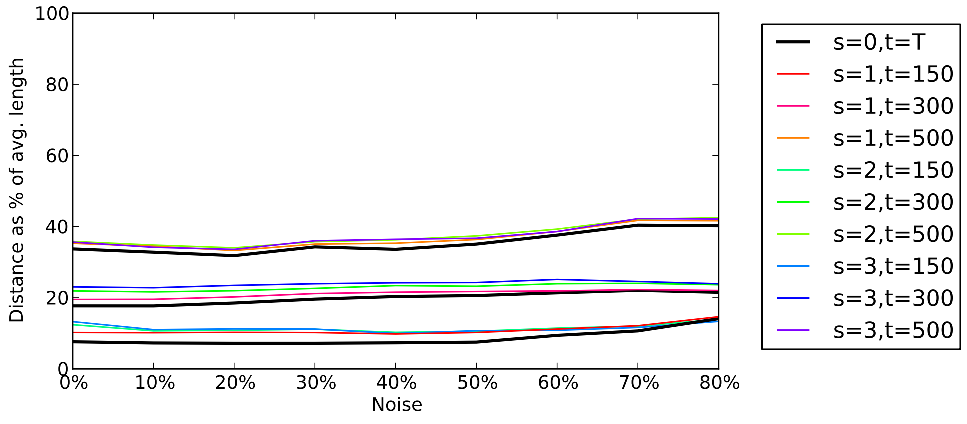


(b) with distance as % of average build order length per race.

Figure B.5: Impact of noise on predictions in Protoss vs Terran.

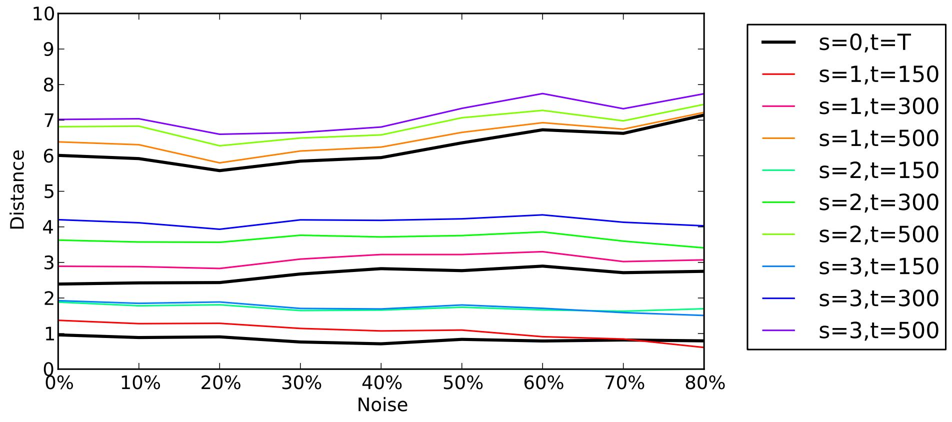


(a) with absolute distance.

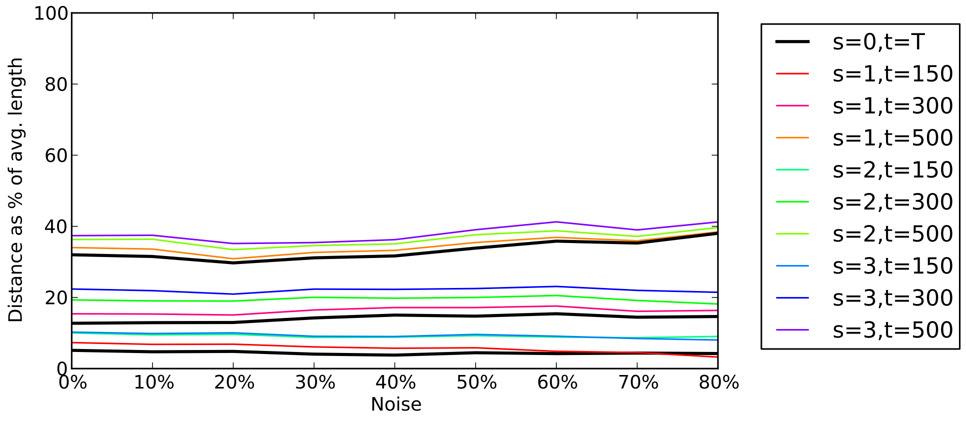


(b) with distance as % of average build order length per race.

Figure B.6: Impact of noise on predictions in Protoss vs Zerg.

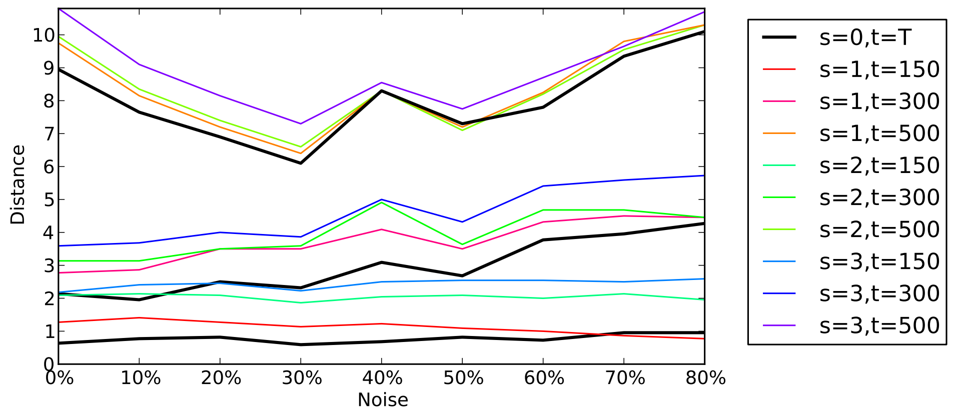


(a) with absolute distance.

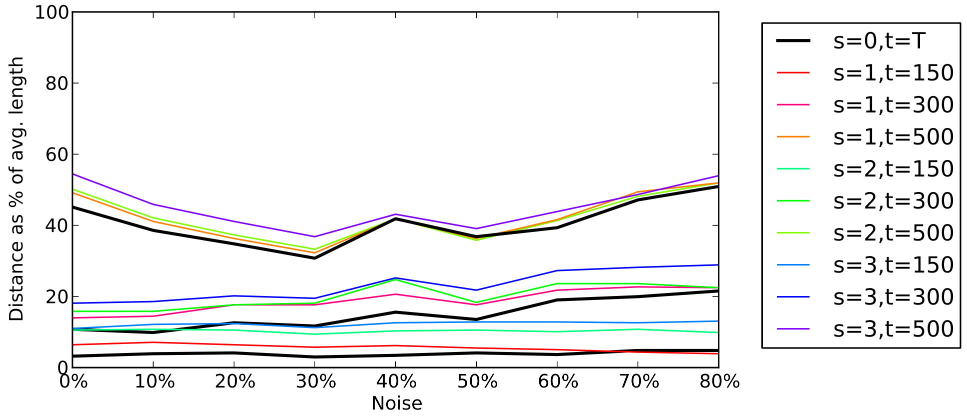


(b) with distance as % of average build order length per race.

Figure B.7: Impact of noise on predictions in Terran vs Protoss.

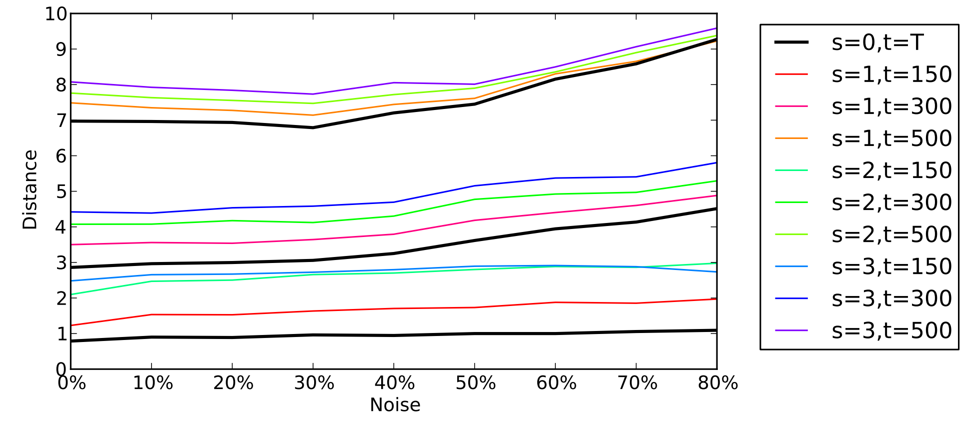


(a) with absolute distance.

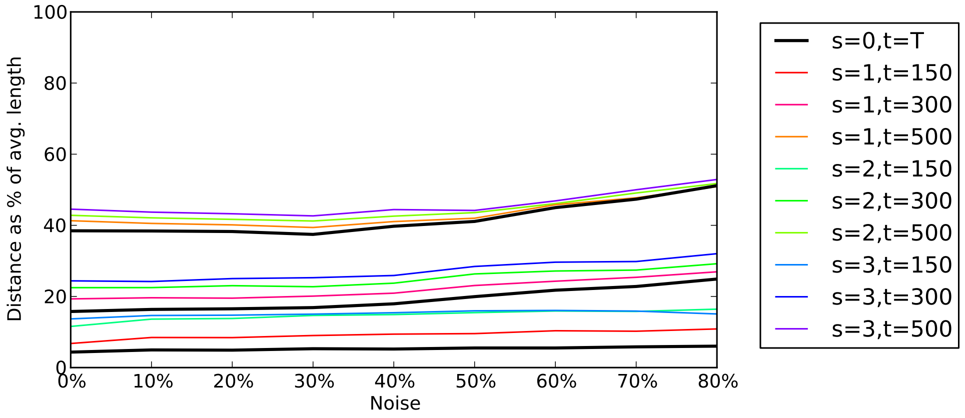


(b) with distance as % of average build order length per race.

Figure B.8: Impact of noise on predictions in Terran vs Terran.

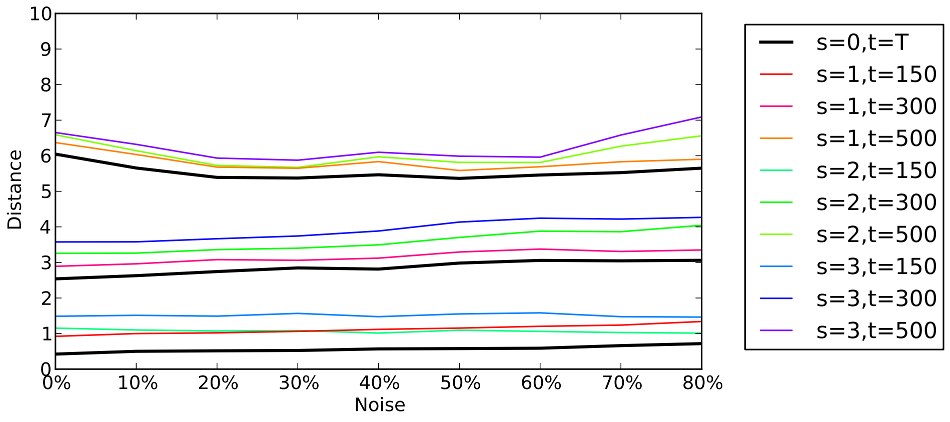


(a) with absolute distance.

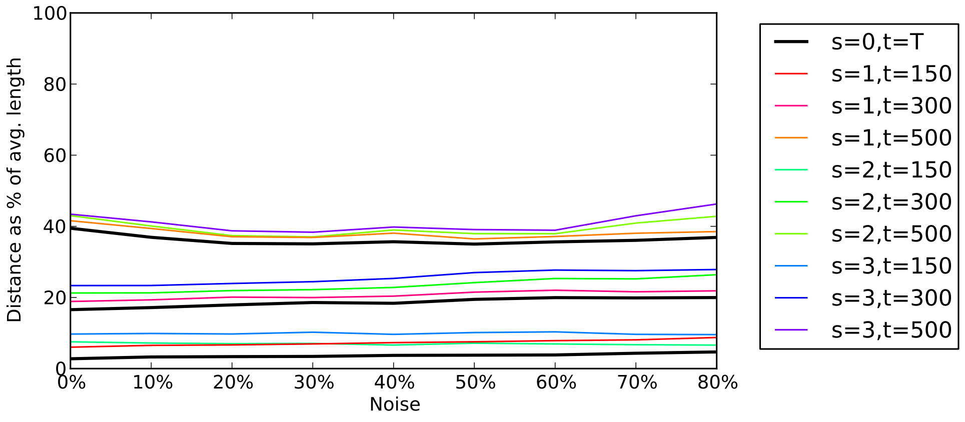


(b) with distance as % of average build order length per race.

Figure B.9: Impact of noise on predictions in Terran vs Zerg.

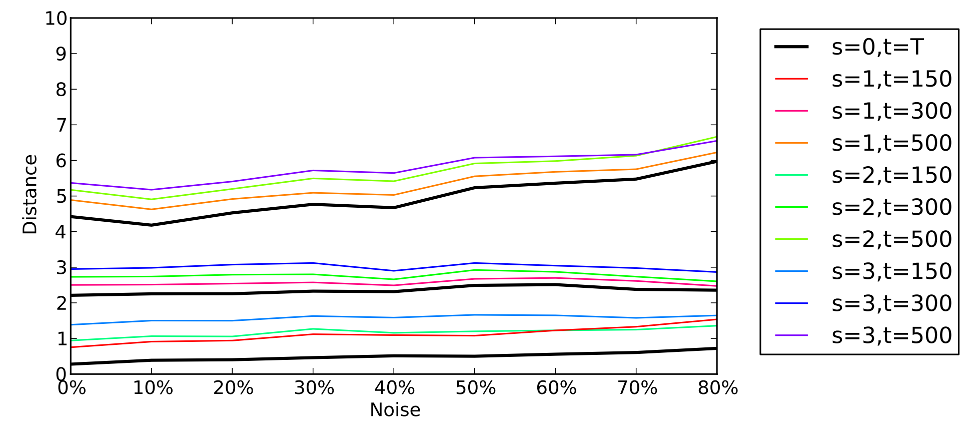


(a) with absolute distance.

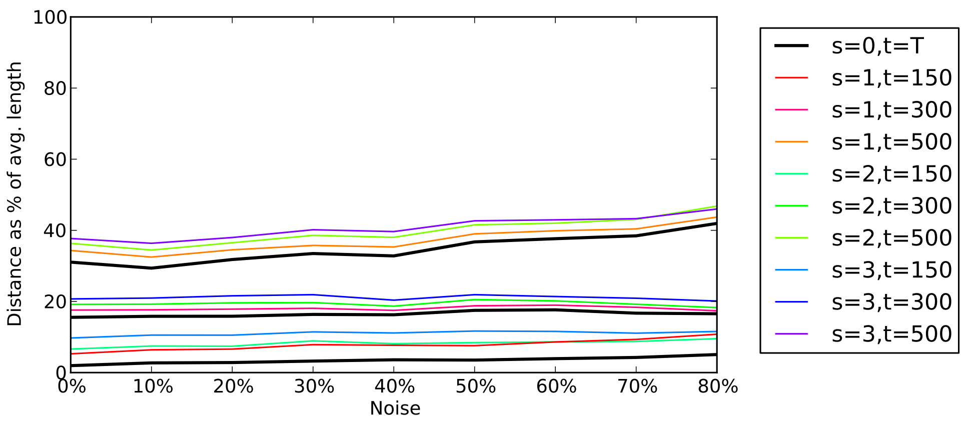


(b) with distance as % of average build order length per race.

Figure B.10: Impact of noise on predictions in Zerg vs Protoss.

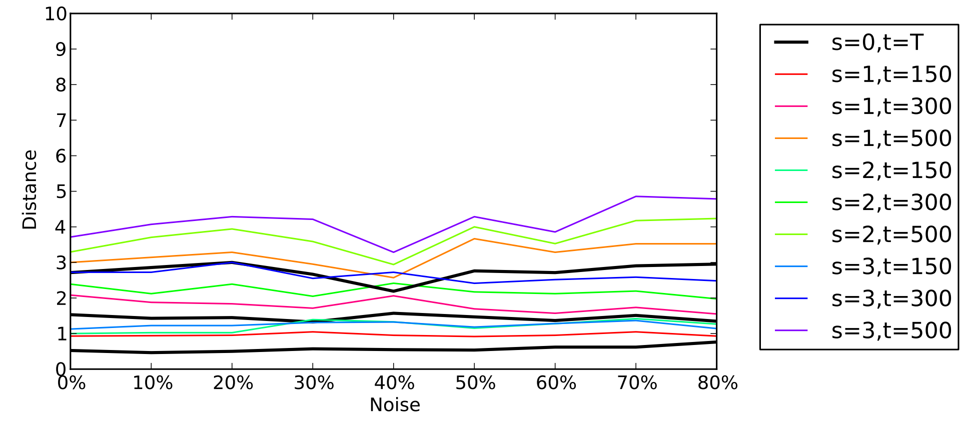


(a) with absolute distance.

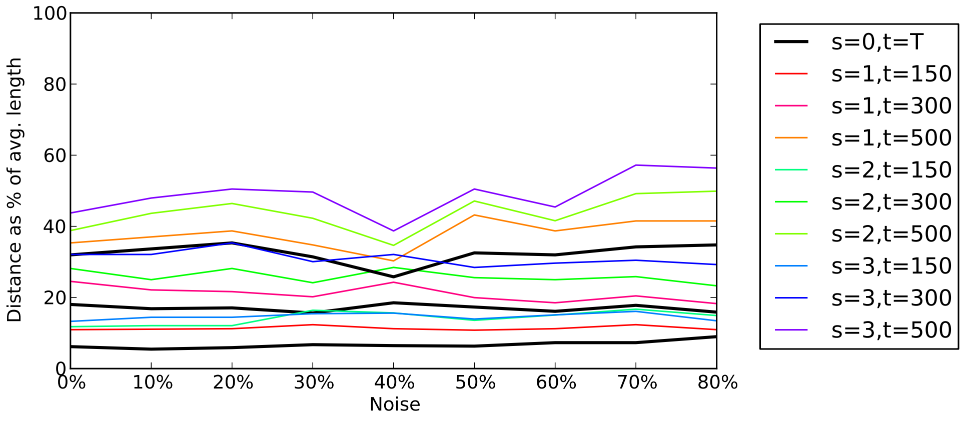


(b) with distance as % of average build order length per race.

Figure B.11: Impact of noise on predictions in Zerg vs Terran.



(a) with absolute distance.



(b) with distance as % of average build order length per race.

Figure B.12: Impact of noise on predictions in Zerg vs Zerg.

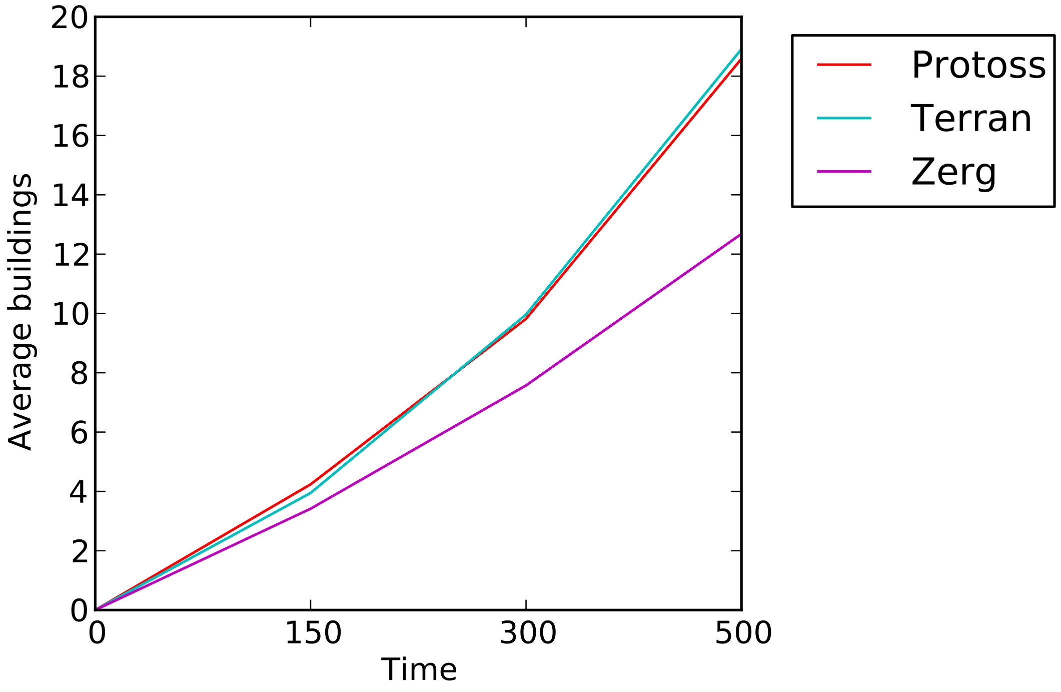
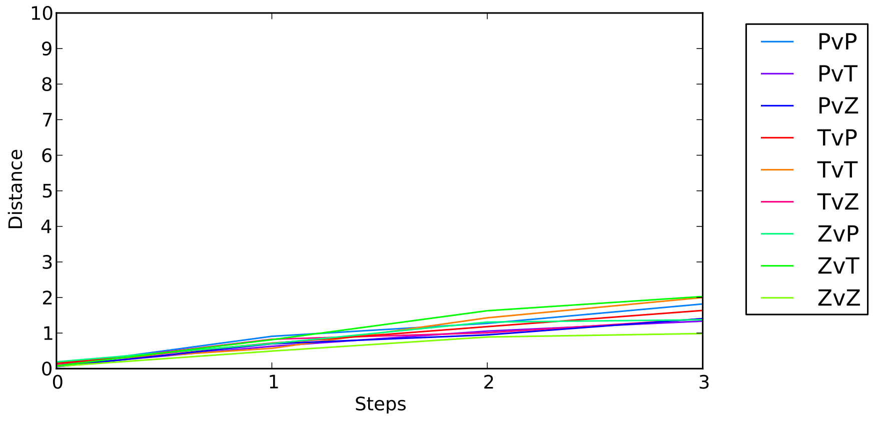


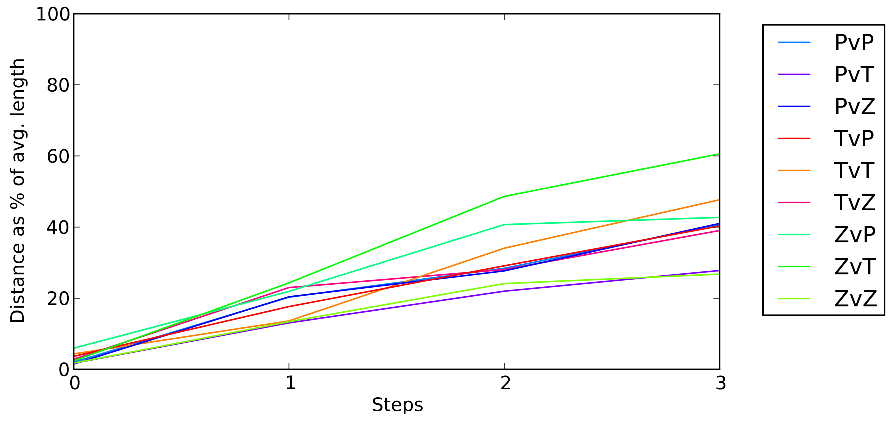
Figure B.13: The average length of the build order per race over time.

B.2 Strategic Reasoning

The graphs created from the results of the experiments on the strategic reasoning model.

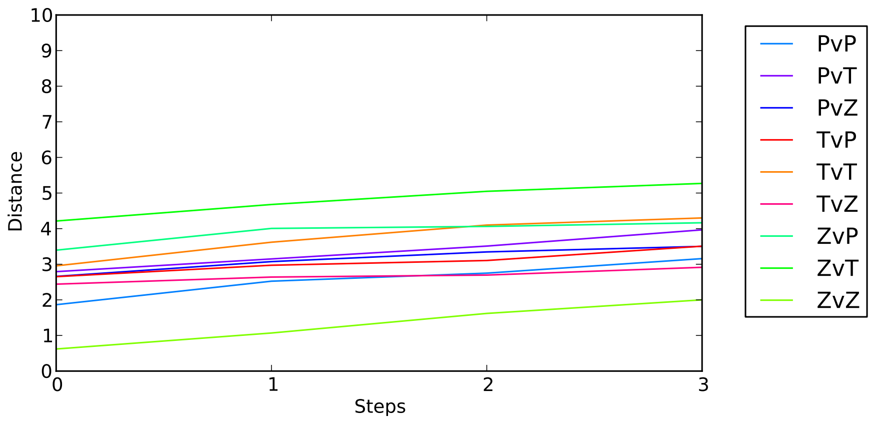


(a) with absolute distance.

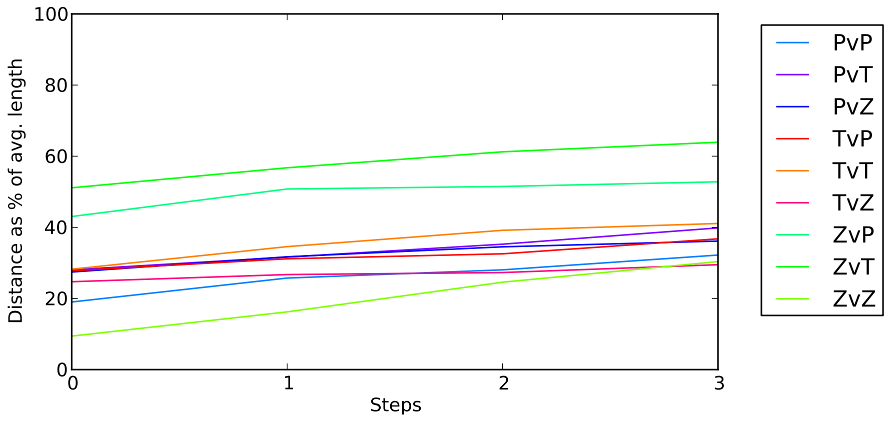


(b) with distance as % of average build order length per race.

Figure B.14: Results of counter tree recognition at $t = 150$.

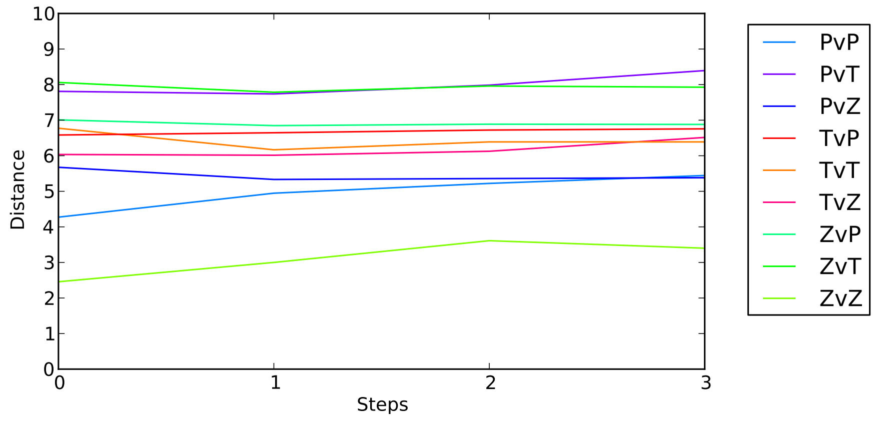


(a) with absolute distance.

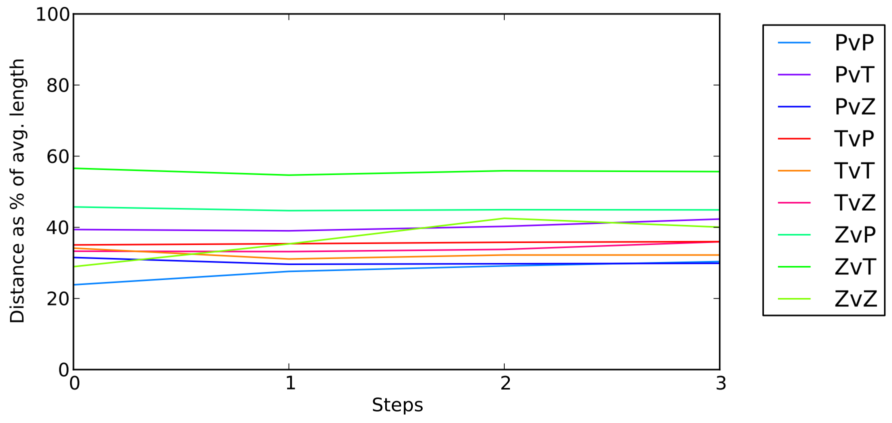


(b) with distance as % of average build order length per race.

Figure B.15: Results of counter tree recognition at $t = 300$.



(a) with absolute distance.



(b) with distance as % of average build order length per race.

Figure B.16: Results of counter tree recognition at $t = 500$.