State Evaluation and Opponent Modelling in Real-Time Strategy Games

by

Graham Erickson

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science
University of Alberta

Abstract

Designing competitive Artificial Intelligence (AI) systems for Real-Time Strategy (RTS) games often requires a large amount of expert knowledge (resulting in hard-coded rules for the AI system to follow). However, aspects of an RTS agent can be learned from human replay data. In this thesis, we present two ways in which information relevant to AI system design can be learned from replays, using the game StarCraft for experimentation. First we examine the problem of constructing build-order game payoff matrices from replay data, by clustering build-orders from real games. Clusters can be regarded as strategies and the resulting matrix can be populated with the results from the replay data. The matrix can be used to both examine the balance of a game and find which strategies are effective against which other strategies. Next we look at state evaluation and opponent modelling. We identify important features for predicting which player will win a given match. Model weights are learned from replays using logistic regression. We also present a metric for estimating player skill, which can be used as features in the predictive model, that is computed using a battle simulation as a baseline to compare player performance against. We test our model on human replay data giving a prediction accuracy of $> 70\%$ in later game states. Additionally, our player skill estimation technique is tested using data from a StarCraft AI system tournament, showing correlation between skill estimates and tournament standings.

Preface

This thesis involves work done for the purposes of publication. Chapter 3 is original work. The work presented in Chapter 4 is being published at AIIDE 2014. I (Graham Erickson) am the primary author and Professor Michael Buro is the other author. Chapter 2 is original work, but is adapted from a literature review done for CMPUT 657.

Acknowledgements

Thanks to my supervisor Michael Buro for guiding me through this thesis and offering invaluable insight. Thanks to the RTS research group and especially David Churchill, Marius Stanescu, and Nicolas Barriga who helped me immensely during my time at the University of Alberta.

I would also like to thank all of my friends (both in Edmonton and Saskatoon) for helping me through tough times and making my experience in Edmonton all the more enjoyable.

I owe a lot of gratitude to my parents (Wendy and Kelly) and my sister April. Their support has been crucial to my success and I would not be where I am today without them.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Purpose

Real-Time Strategy (RTS) is a genre of video game in which players compete against each other to gather resources, build armies and structures, and ultimately defeat each other in combat. RTS games provide an interesting domain for Artificial Intelligence (AI) research because they combine several difficult areas for computational intelligence and are implementations of dynamic, adversarial systems [1]. The research community is currently focusing on developing AI systems to play against each other, since RTS AI still preforms quite poorly against human players [2]. The RTS game StarCraft (en.wikipedia.org/wiki/StarCraft) is currently the most common game used by the research community, and is chosen for this work because of the online availability of replay files and the open-source interface BWAPI (code.google.com/p/bwapi).

This thesis combines two distinct projects (which are related thematically). The first deals with the abstract notion of *strategy*. In common language, strategy can be viewed as a high-level plan (or abstraction of a plan) that can be implemented to achieve a goal. In RTS games strategies are often viewed as general rules that characterize a way of playing the game (e.g. sacrificing economy to gain an early military advantage is called a *rushing* strategy). In this thesis when we discuss strategy we refer to pure strategies (in the game theoretic sense). Humans players often have a few different strategies which they implement during matches and typically have a good sense for which strategies are effective against other strategies. Having such knowledge requires in-depth experience with a game, and using human opinion as a basis for building strategy into an AI system introduces bias and removes the possibility of novel strategies to emerge. The purpose of part of

this thesis is to provide an empirical basis for identifying strategies and discovering inter-strategy strengths and weaknesses.

The second project concerns the value (another abstract concept) of states in RTS. When human players are playing RTS games, they have a sense of when they are winning or losing the game. Certain aspects of the game which can be observed by the player are used to tell players if they are ahead or behind the other player. The goal of a match is to get the other player to give up or to destroy all that player's units and structures, and achieving that includes but isn't limited to having a steady income of resources, building a large and diverse army, controlling the map, and outperforming the other player in combat. Human players have a good sense of how such features contribute to their chances of winning the game, and will adjust their strategies accordingly. They also are adept at determining the skill of their opponent, based on decisions the other player made and their proficiency at combat. We want to enable an AI system to do similar. The purpose of our work is to identify quantifiable aspects of a game which can be used to determine 1) if a particular game-state is advantageous to the player or not; and 2) the relative skill level of the opponent.

1.2 Motivation

The most successful RTS AI systems still use hard coded rules as parts of their decision making processes [3]. Which policies are used can be determined by making the system aware of certain aspects of the opponent. For example, if you have determined that the opponent is implementing strategy A, and you have previously determined that strategy B is a good response to A, then you can start executing strategy B [4]. Knowing that strategy B is effective against A, however, merely comes from expert knowledge, which can often overlook novel relationships between strategies. Having an empirical basis for which strategies are strong against which other strategies also gives game designers a way of analyzing the balance of their game. Finding groupings of like strategies automatically from data would allow game designers to automate game balance detection processes and simplify the development of RTS games. Polishing RTS games is a very complex process, as seen by the length of time that it took to fine-tune StarCraft (the game was still receiving patches up until 2009).

Search algorithms have been used successfully to play the combat aspect of RTS games [5]. Classical tree search algorithms (excluding Monte Carlo based methods) require some sort of evaluation technique; that is, search algorithms require an efficient way of determining if a state is advantageous for the player or not. Currently, there is work being done to create a tree search framework that can be used for playing full RTS game [6]. Evaluation can be done via simulation [7] for combat, but for the full game different techniques will be needed. Also, in the context of a complex search framework that uses simulations, state evaluation could be used to prune search branches which are considered strongly disadvantageous. As we will show in Chapter 4, the type of evaluation we are proposing can be computed much faster than performing a game simulation.

Most RTS AI systems still use hard-coded rules to make decisions, but some are starting to incorporate more sophisticated methods into their decision making process. For example, UAlbertaBot (code.google.com/p/ualbertabot), which won last year's AIIDE StarCraft AI competition, currently uses simulation results to determine if it should engage the opponent in combat scenarios or not. This is based on the assumption that the opponent is proficient at the combat portion of StarCraft. If there is evidence that the opponent is not skilled at combat, one might be willing to engage the opponent even when their army composition is superior (or if they are strong, not engage the opponent unless the player has a large army composition advantage).

1.3 Objectives

The main objective for this thesis is to provide insight into two machine learning problems which have not been acknowledged in the RTS literature. Regarding the strategy clustering problem, we provide a clear method for identifying groups of strategies from RTS replay data and provide our findings on real data. Our method uses agglomerative hierarchical clustering to cluster strategies. We also provide a method for developing distance functions between strategies, which borrows from sequence alignment techniques mostly used in the field of bio-informatics.

We attempt to solve the result prediction problem by presenting a model for evaluating RTS game states. More specifically, we are providing a possible solution to the game result prediction problem: given a game state predict which player will

go on to win the game. Our model uses logistic regression to give a response or probability of the player winning (which can be looked at as a value of the state for the player). Presenting our model will then come down to describing the features we compute from a given game state. The features come in two distinct types: features that represent attributes of the state itself (which can be correlated with win status), and features which represent the players skill (which is a much more abstract notion). Our model assumes perfect information; StarCraft is an imperfect information game, but for the purposes of preliminary investigation we assume that the complete game-state is known.

1.4 Contributions

This thesis contains three main contributions to the field of RTS AI. The first is a technique for clustering build-orders. This allows a researcher to group build-orders (found in a data-set of replays of a RTS game). The point of this is get a sense of what kinds of build-orders players are generally using. The benefit to this technique is that novel build-order groupings can emerge from the data-set and it removes the need for advanced expert knowledge when choosing strategies for an AI system to implement. The clusterings can be used to build and populate payoff matrices using the match outcomes from the replay data-set. These payoff matrices can be used to gain insight into which types of build-orders tend to beat which other types of build-orders in real games. Such information can be considered when designing AI systems, in terms of response strategies. Payoff matrices have also seen to be useful for analyzing the balance of a game. Our process can be used to automate balance detection (which is very important when developing commercial RTS games).

Predicting the result of an RTS match is a noisy problem. There are many factors that contribute to a player winning or losing a match and key moments can quickly shift the momentum of a game. In this thesis, we provide a set of features that can be used to predict the outcome of a game (i.e. which player will win) with fairly decent accuracy ($> 70\%$ in later game stages). AI systems can use our feature set for state evaluation both to prune nodes in a global search and to inform decision making. Our feature set also reveals which features are most important to the outcome of a match. Future works could focus on having systems try to improve

the values of certain features when in losing situations in a game. We also provide a metric for estimating the skill of a player at the level of micro-managing units in a battle. Micro skill is considered an important part of playing RTS games well. Our metric provides an empirical basis for estimating the micro skill of a player. This technique can be used to model our adept an opponent is at managing units in battle, which can influence decision making (an AI system could be more aggressive against subpar opponents and defensive against competent opponents). Our method can also be used to add information to player rankings or to allow players to have a metric for quantifying how proficient they are at battle management (in case they need to improve).

1.5 Contents

The next chapter presents a brief literature survey of RTS games. In Chapter 3 we describe the build-order clustering scheme and show how it can be applied to StarCraft. Then in Chapter 4 we explore the result prediction problem, present our feature set, along with the battle skill estimator, and show our experiments with real data. The strengths and weaknesses of our model, along with future plans are discussed in Chapter 5.

Chapter 2

Background

Research into RTS is a growing field and before presenting the different works that have been done, we will briefly explain the different games which are commonly used as experimental domains. One of the first games used to research RTS is called *ORTS* (open real-time strategy) [8]. ORTS is an open-source RTS engine that allows researchers to create games that are particular to a use or a purpose. It is designed to be easy to use, and is open-source so there are no problems with interacting with an obfuscated game system (which can be a problem when trying to develop AI system to play a commercial game). *Wargus* has also seen some use in the research community [9]. Wargus is a clone of the older RTS game *WarCraft II*. Currently, StarCraft is the most popular game for RTS research. StarCraft was a very commercially successful game and has many replay files freely available online. StarCraft is known to be a very well-balanced game and has three different factions (called *Protoss*, *Zerg*, and *Terran*) which benefit from varying play mechanics. For RTS AI research, StarCraft can be interacted with using BWAPI (Brood War API) and AI system development competitions using BWAPI have shown to be popular and interesting ways of promoting and testing RTS research [2].

2.1 Search in Real-Time Strategy Games

Search algorithms have a long history in classic game playing. Minimax search using Alpha-Beta pruning has had great success in games like chess and checkers [10], and the technique has been given modifications that have proved successful in games like Othello [11]. Chess and checkers are perfect information games and have sequential moves (which make them simpler games to adapt minimax to, as opposed to imperfect information RTS games which feature simultaneous moves).

RTS games also have extremely large branching factors and there are many different ways to play a game successfully. Consider the amount of moves available to players at any one time; players can build units and buildings and command any of potentially hundreds of units. Couple the amount of moves available with problems in temporal and spatial reasoning and RTS games appear to be a very difficult domain for tree search algorithms to play. For large domains, an evaluation function is required (i.e. a method for telling how advantageous a state is for the player), since a search cannot be done on the complete tree in a reasonable amount of time.

More recently, Monte Carlo search techniques have seen success in games with large branching factors, like Go [12]. Monte Carlo search is stochastic in nature (which is different from Alpha-Beta search, which is deterministic) and has been applied to non-deterministic, imperfect information games like Poker [13]. Monte Carlo search focuses on simulating full play-outs of a game and collecting statistics regarding which moves tend to lead toward victories for the player. Both Monte Carlo techniques and Alpha-Beta search have seen applications in RTS games.

MCPlan is a Monte Carlo style planning algorithm which was developed and implemented for a capture-the-flag style game in ORTS [14]. MCPlan incorporates both abstractions and random sampling. In a general sense, MCPlan works by randomly generating plans for both the player and opponents. The results of the plan for the player are recorded, and the process is repeated for as long as time constraints allow. Then the player actually executes the plan which had the most statistically significant success during the random play-outs. In implementing MCPlan for the capture-the-flag game, an evaluation function is needed (i.e. a way of measuring the success of a play-out is needed, since in this case play-outs are not done to a terminal state). The authors use a combination of material evaluation (units are weighted based on their health, and material is a sum of the player weights subtracted by the opponent weights), visibility evaluation (value is given to plans that explore and reveal the map), and flag capture evaluation (plans are rewarded for player proximity to the opponent flag and punished for opponent proximity to the player flag). It should be noted that the parameters for each evaluation scheme were tuned manually, instead of learned from data. The evaluation function is then a weighted sum of the three evaluation schemes.

Monte Carlo tree search has also been applied to Wargus, for planning at the

tactical level [15]. In the paper, UCT (a Monte Carlo style algorithm that has had great success in Go [16]) is adapted to what they call the tactical assault problem (i.e. the shooting game in which each player has a certain number of units, and the AI player seeks to defeat all the enemy units while maximizing the leftover health of the player units). The state space of even just the tactical assault portion of the game is very complex (PSPACE-hard to be exact [17]). To compensate, an abstract version of the game is used. Groups of units are reasoned with instead of just individual units (groups are made based on spatial proximity). So the planning is done using properties of unit sets, and the primary abstract actions are to join groups and to attack groups. Also, the paper notes that the work done in ORTS [14] relies on a good evaluation function, which might not be easily developed and adapted for different applications, and that the work here differentiates itself because the UCT play-outs go to the end of the tactical assault matches (and thus do not require intermediate evaluation) and that the tactical assault scenario is more general than the capture-the-flag scenario. The UCT algorithm is implemented as part of an online planner. At certain time steps, known as decision epochs, the units are clustered to form abstract unit groups and the UCT algorithm is ran on the unit groups. Then the actions that the algorithm decides upon are ran until the next decision epoch, when the whole process is repeated. For the actual search, states (nodes) are a set of groups of units (each having a collective health and position), a set of actions given to the unit groups, and a time stamp. Arcs are actions given to a group of units.

Alpha-Beta search has been shown to be useful for playing RTS games at the micro level [7]. Combat scenarios can be modeled as an individual game (or rather a sub-game), where two players controlling a fixed number of units must try to defeat the opponent's units while maximizing their unit's left-over health. Since StarCraft is very complex, an abstract model of the combat game is required for search purposes. The abstract game works on sets of units and moves apply to sets of units as well. To simplify the problem, many complex aspects of StarCraft are ignored (spell-casters, hit-point regeneration, imperfect information and unit collisions). Levels in the game tree which represent simultaneous moves in the abstract game can be replaced with two levels representing alternating player moves. Evaluation is used as part of the search. A very useful evaluation function in this work is a sum of the square-root of player unit hit-points (the square-root smooths out

the hp distribution), weighted by a ratio that describes the rate at which units can deal damage (which offers a very fast form of evaluation). Evaluation is also done using scripts, which deterministically play the game from a given state (using a heuristic). Script-based evaluation is slower than using weighted sums, but allows evaluation to be done in terms of terminal play-outs. A search method has also been built for the combat game that searches over a set of possible scripts [5].

Work is currently being done to develop search algorithms that can be applied to a higher level of RTS game (instead of simplified combat). This work is currently in preliminary stages [6], but the general idea is to create a hierarchy of abstract searches that take advantage of solutions to sub-problems. Although research has not reached the level of a search algorithm that plays over states that encompass the entire game, work into hierarchical search methods show promise that such a search algorithm exists. As part of the search process, intermediate (but global) states would be searched over. Global evaluation could benefit such searches immensely, by allowing intermediate states to be pruned when the evaluation shows that the states are much worse than others (for the player).

2.2 Machine Learning in Real-Time Strategy Games

Techniques have been used in developing RTS AI that use data to develop decision-making models, or to give insight into the game itself. Machine learning is often used to predict the opponent's actions or model the opponent in some way. Opponent modelling in RTS was first done using an RTS engine called SPRING, and did not use machine learning at all [18]. Instead it used an expertly designed fuzzy logic system for opponent strategy identification. Replay data was used soon after for modelling how RTS games can be played [19]. This work was done before BWAPI existed however, and never saw use in the context of an RTS AI system.

One trend that can be seen in the RTS AI literature is the application of Case-Based Reasoning (CBR) techniques [20] [21] [22]. In general, the idea is to identify particular cases where a certain tactic or strategy should be used. The area is a combination of machine learning and planning. The approach starts with a set of previous experiences (also called cases). Then in live play, the system selects counter strategies from the previous cases and applies them to the current situation. Cases are selected based on their similarity to the current situation. The results

are then used to update the previous cases. CBR using fuzzy set logic has also been applied to StarCraft, with success against the built-in StarCraft AI system [23]. It should be noted that the built-in StarCraft AI system is quite simple and is well-known to be not particularly good at playing RTS. A similar concept know as transfer learning has been applied to an RTS game called MadRTS [24]. Previous experiences take the form of plans, are applied when applicable and are evaluated for further use depending on the outcome.

Currently, many of the competitive AI systems use models learned from replay data in some way. The trend can be traced to Weber and Mateas' work in 2009 [25], which is one of the earliest examples of using machine learning on StarCraft replay data to develop an opponent model and applying the resulting to model to a StarCraft playing AI system. A player's strategy is considered to be a generalization of a player's build-order. The problem the paper is concerned with is how to detect what strategy the opponent is executing given some evidence about the opponent. Human players in RTS games are often concerned with trying to figure out what strategy the opponent is executing, so that the player can try to execute a counter-strategy. RTS games are imperfect information games and most of the opponent's actions (especially early on in a game) are hidden from the player. In order to get hints at what sort of strategy the opponent is executing, players must scout (by sending units into unknown areas of the map purely with the intention of gathering information). When a player sees what sort of units and structures the opponent has built, they can make an educated guess at what sort of strategy the opponent is executing (based on past experiences). Analogously, when a system gathers evidence about the opponent by scouting, the system can refer to models developed on replay data (which can be seen as past experiences) to guess at what strategy the opponent is executing. In [25], vectors were extracted from replay files that have a feature for each unit or structure type. The value for the feature is the time in a match that the player in the replay first produced a unit or structure of that unit or structure type. The vectors were labeled with the names of high-level strategies (assigned by a set of rules). Ten-fold cross-validation was run using a few different machine learning algorithms. Logistic regression with boosting was found to be the most effective at predicting the strategy labels from the vectors. Our work does not deal with strategy prediction, but borrows the idea of strategies and build-orders being analogous and uses replay data to develop models.

StarCraft has the built-in functionality to record a match and save it in a specific binary format that can then be reinterpreted by the game engine (for the purpose of replaying the match using the StarCraft software). Communities have developed around the web where amateur players can post match replays and where the replays of top matches on amateur competitive brackets are posted. Replays can then be downloaded and parsed to extract the relevant information. Parsing replay files requires either loading the replay into StarCraft and extracting the desired information using BWAPI and an AI system, or using some sort of proprietary software tool to parse the raw StarCraft replay data.

There has also been some work that uses probabilistic graphical models as part of the opponent modelling process. Hidden Markov models (HMMs) have been used as part of a system to detect opponent behaviours in the form of plans [26]. They have also been used to actually learn the strategies themselves from data as well [4]. The advantage of this approach is that strategies aren't pre-determined by experts, which allows the emergence of novel strategies and gives an empirical basis for strategy specifications (i.e. when labeling feature sets manually, a human may inject biases or inconsistencies). Games are split up into thirty second intervals (the states in the HMM). Each interval is given a vector that has a feature for each unit/structure type (the observations in the HMM). A few hundred replays of Protoss players facing Terran opponents were gathered and expectation maximization was used to learn the model. After the state model was learned, the authors graphed the states as nodes and drew arcs between one node and another if the first node's state has a non-zero probability of transitioning into the other node's state. A path through the resulting state transition graph (including loops) is understood to be a strategy which represents the player's behaviour throughout a match. The interesting thing is that strategies which are well known by the community emerged from the data and can be seen quite clearly in the state transition graph. The work is a primary example of how data analysis of human replays can be used to learn information about the game itself.

Bayesian models can be used to model player behaviour (for various purposes), as shown in the work of Synnaeve. In [27] and [28] a Bayesian model is described that can be used to predict opponent opening strategies and build orders. Here games are represented as feature sets (representing when a unit/structure started to be produced) and each feature set is given a label that describes the strategy

being used (the work here is concerned primarily with identifying the opening strategy of a player). A difference between Synnaeve's strategy labeling and that of Weber and Mateas, is that here labeling is done using a semi-supervised method. Clustering is used to identify strategy groups and those are manually given labels (as opposed to simply giving each feature set a label manually or via a set of rules). Clustering is done on the feature sets and not the build-order sequences themselves. The data is used to learn parameters for a Bayesian model, which can be used to predict opponent strategies given observations (like seen units). The performance of the model is compared as a classifier against the performance of Weber and Mateas' model (which isn't a completely fair comparison since they use different labellings of the data). Synnaeve's model is found to be slightly less accurate overall (although way more accurate for some faction match-ups). Their model is considered by the authors to be quite robust to noise, and since the the model is probabilistic, uncertainty is quantified as part of the model itself. Similar models have been developed for making tactical decisions [29] and controlling units at a lower level of abstraction [30].

A problem facing researchers experimenting with learning models from replay data is that up until recently, a large general easily-usable data-set did not exist. The data used by Weber and Mateas can be obtained from them, but the information about each match only contains what is relevant to their work. If a researcher wanted to analyze replay data for other purposes, they would have to scour the web looking for various replay files on matches between experienced players. Synnaeve et al. performed the collection and formatting of a large, general data-set for StarCraft AI research [31]. The authors collected nearly eight-thousand replay files on one-versus-one StarCraft matches between experienced players, and used BWAPI to gather a large amount of data about each match. The collected data was then written to text files. The work done makes the job much easier for future researchers, who can now bypass the data collection and extraction phases, and simply parse Synnaeve's text files to mold the data into the desired format. The data parsed from the replays includes all observable player actions, a running count of both player's resources (dumped every twenty-five frames or approximately every second), times for when units are seen by the various players (to incorporate fog-of-war), and the effects and timing of attacks executed by all units. We use the dataset for the projects presented in this thesis. We use the parsed files for

the work done in Chapter 3, but we opted to build our own parser for Chapter 4 because we wanted complete control over the information we gathered (e.g. we redefined what constitutes a battle). Synnaeve also describes an experiment in unit clustering as an example for how the dataset could be used. The clustering is done on "army" compositions (groups of units that engage in battle) so it differs from our clustering project.

There have been a few other modern examples of learning and probabilistic modelling in RTS games. Weber et al. dealt with the uncertainty caused by imperfect information using a particle filter to predict unit positions in fog-of-war [32]. Reinforcement learning has been used to develop micro-management techniques for small combat scenarios [33]. The model works with a simplified version of a StarCraft battle, where units are allowed to either attack or retreat. The learner then rewards or punishes the AI system after each decision and the system changes its decision-making process accordingly. Gemine et al. looked at replay data from StarCraft II to genetically develop production policies (rules for different unit types about when they should be produced) [34]. Evolutionary computation has been used to improve the tactical decision-making of a StarCraft AI system [35]. A combination of evolutionary computation and a neural net was used to teach a program to play Wargus [36].

As far as we can tell, little to no work has been done in predicting game outcome. [37] tries to predict game outcomes in Massively Online Battle Arena (MOBA) games, a different but similar genre of game. They represent battles as graphs and extract patterns that they use to make decisions about which team will win. Bayesian techniques have seen success in predicting the outcome of individual battles, using data from a simulator [38]. That work focused just on individual skirmishes and did not include the whole match. [39] extracted features from StarCraft II replays and showed that they can be used to predict the league a player is in.

2.3 Replay Data for Building Payoff Matrices

The project described in Chapter 3 is largely an extension of part of the work described in Long's Master's thesis [40]. In that work, game theoretic definitions concerning the balance of a game are established. Balance can mean either that there

is no faction that isn't useful in some situation or that there is no strategy that isn't useful in some situation (this is a simplified definition, but it captures the intuition, which is acceptable for our purposes). Long proposes building payoff matrices from replay data to analyze a game for balance. The idea is that for a particular faction match-up, game replays from human matches can be used to populate a payoff matrix. The rows and columns of the matrix represent different strategy choices for the two factions. The thesis presents a study in which 100 WarCraft III replays are hand labeled by expert observers (labels are high-level descriptions of the strategy used). Strategy here corresponds to the build-order used (the order units are produced in). The results of the game replay can then be used to populate a payoff matrix to check if the game is balanced. We are interested in discovering rock-paper-scissors patterns; matrices that show that strategies have other strategies they are strong against and others they are weak against. Our work differs from Long's because we do not label replays, and instead use clustering to identify natural strategic groupings in replay data. We also use significantly larger datasets.

Long's thesis also uses the labeled replays data to stage a machine learning problem. He models the strategy labels as target values, and the build-order sequences as the examples. A model can then be learned with predicts the strategy label given a build-order. This work suggests a method for determining the distance between one build-order to another (distance here is a measure of how similar or different two build orders are, and is used in learning the predictive models). The method borrows from the field of bio-informatics, which has long used sequence alignment techniques to make sense of large amounts of data in the form of sequences. Long uses alignment scores as distances between build-orders. We use a similar approach to develop a similarity function between build-orders that is used in the clustering process. More details are given in Chapter 3.

2.4 SparCraft

SparCraft is an open-source StarCraft battle simulator developed by Churchill [41]. StarCraft is a complex piece of software and because it is not open-source, it must be treated as a black box. This can cause complications when trying to develop more sophisticated algorithms for StarCraft AI (such as search) [42]. Also, when running searches in a game, it is useful to have a general and abstract version of

the RTS game that can be used to perform play-outs from various states (this is not yet feasible for the entire game but can be done for sub-problems). SparCraft was developed as general StarCraft combat simulator that could be used as both for experimenting in a simplified (but still StarCraft applicable) environment and as a tool for use during a game (either as part of a search or for other forms of decision making). We use SparCraft in Chapter 4 as part of a method for determining a player's skill at the combat portion of the game.

SparCraft makes several simplifications of the full StarCraft game. Spell-casters are ignored (expect for Terran medics) because of their diversity and complexity. Flying units are not allowed (also for simplicity). Collisions between units are ignored (collisions do not affect a battle significantly). Projectile attacks happen instantaneously. We modified SparCraft so that it allowed for buildings (with collisions) to be included, and allowed units to enter battles are varying times (since in a real match players often reinforce their in-battle units with additional units).

2.5 Baseline

As presented in [43], a *control variate* is a way of reducing the variance in an estimate of a random variable. The authors apply control variates (in conjunction with a baseline scripted player) to Poker, as a way of estimating a player's skill. The main idea, that is relevant to our work, is to use a scripted (or simply computationally less complex) player to provide a comparison against which to consider the performance of an agent. The scripted player plays out the same scenario which the agent encountered and both performances are evaluated. The two values can then be compared to give an empirical measure of the skill of the agent. We apply the idea to the combat portion of StarCraft. We use a StarCraft combat simulator (SparCraft) to replay battles with a baseline player, and the control variate technique to reduce the variance of the resulting skill feature estimate. More details are given in Chapter 4, where the resulting skill estimate is used as part of our feature set for the game result prediction problem.

Chapter 3

Build-Order Clustering

The work done in Long's thesis leaves an interesting possible extension: instead of hand-labeling build-orders with strategy labels, use clustering techniques to identify groups of build-orders that embody similar strategies. In this Chapter we describe a general process for representing and clustering build-orders to identify groupings in a dataset of game replays. We also show how the general process can be adapted for a particular game, using the RTS game StarCraft.

3.1 Representing Strategies

Recall that *strategy* refers to the highest level of decision making. *Strategy* can be seen as more long-term planning, in the sense that strategic plans tend to characterize a whole game (or at least a significant portion of a game). However, strategy does not refer to a specific single thing and is a combination of aspects of high-level decision making. In order to quantify strategy a suitable *abstraction* is needed. Choosing a good abstraction comes down to choosing which quantifiable aspects of a player's decisions best capture strategy as a whole.

For a particular match, the high-level plan followed by a player is a strategy. Much like Jeff Long [40], we choose to represent strategy in terms of *build-order*. A build-order is the order that units and structures are built by a single player in a game [44]. Build-orders are suitable stand-ins for the abstract concept of strategy because the essence of high-level strategy is the existence of certain units, and the order units are built in reflects the other more abstract aspects of strategy (e.g. lots of military units early on represent rushing strategies, build-orders dominated by flying units correspond to an air-based assault, etc.).

Build-orders are sequences, where the elements in the sequence represent a

corresponding unit or structure being built. Thus we can encode build-orders as strings. Each of the available units and structures in the game is assigned a unique character. The order that characters appear in a build-order string corresponds to the order that the corresponding units or structures were built in the game.

3.2 Similarity Matrices

We wish to cluster strategies, so since we are representing strategies with build-orders, we need a way of clustering build-orders. Since build-orders are not vectors, common clustering methods such as k-means (which require both distance metrics and a way to compute the mean of a group of elements) will not work. We propose first creating a *similarity matrix*, and then clustering build-orders based on the contents of the similarity matrix. A similarity matrix is a matrix which contains pair-wise similarity scores for a set of elements. For our case, the rows and columns represent build-orders and the contents represent how similar corresponding build-orders are. For a similarity matrix S, for build-orders at row i and column j (i can equal j), the similarity score between the build-orders represented at i and j is S_{ij}. The similarity score itself is a function of two build-orders which results in a real number. In general, higher values mean two build-orders are more similar and lower values mean they are more dis-similar.

3.2.1 Sequence Alignment

To populate our similarity matrix, we need an appropriate function of how similar two build orders are. Since build-orders are sequences, we can examine the concept of *sequence alignment* as a way of developing a similarity score. Sequence alignment can be used as a measure of how similar two sequences are [40]. Sequence alignment is mostly studied in the area of bio-informatics [45], but has also been applied to other domains, such as natural language processing [46] and transactional data mining [47]. In general, sequence alignment is the task of identifying similar patterns between sequences. Alignments can be done over complete sequences (global alignment) or just with parts of sequences (local alignment). For the purposes of this thesis, when we refer to sequence alignment, we are referring to the problem of global sequence alignment, as described by Needleman and Wunsch [48].

The basic problem is given two sequences, at what places in the sequences

should "gaps" be inserted in either sequence in order to maximize the similarity (alignment score) between them. Take $S(a, b)$ to be the similarity between two characters a and b, and take $S(-, a)$ to be the *gap penalty* for some character a. Typically, S is chosen so that scores of the form $S(a, a)$ are positive integers and scores of the form $S(a, b)$ with $a \neq b$ are negative integers (but not necessarily). Two sequences do not need to be the same length to be aligned, but will be the same length after they are aligned. Let A and B be two unaligned sequences, and let A' and B' be the aligned versions of A and B respectively. The length of A' and B' is n. The alignment score between A and B is then:

$$\sum_{i=0}^{n} S(A'_i, B'_i)$$

The Needleman-Wunsch algorithm itself maximizes the alignment score. For example, if the two sequences are **abba** and **ba** and S is

$$S(a, b) = \begin{cases} 0 & \text{if } a = b \\ -1 & \text{if } a \neq b \end{cases}$$

a resulting alignment is

abba

_b_a

which has an alignment score of -2. When 0 is used for a match and -1 for a gap or a mis-match the resulting alignment score is equivalent to a commonly used string distance metric called the Levenshtein or edit distance [49].

The Needleman-Wunsch sequence alignment algorithm is a dynamic program that follows a greedy approach. Let n and m be the lengths of sequences A and B respectively. The algorithm fills in a matrix M that is n-by-m. The idea is that row i in M represents the i-th character in A and column j in M represents the j-th character in B. The entry at M_{ij} is the score of an optimal alignment between the first i characters in A and the first j characters in B.

To compute M, first the 0-th row and 0-th column must be filled in. The column at index 0 contains the alignment scores for the characters up to and including i in A being aligned with an empty string (so every character is matched with a gap). Likewise, the row at index 0 contains the alignment scores for the characters up to and including j in B being aligned with an empty string. Algorithm 1 shows how this part of M is initialized.

Algorithm 1 Initializing M

$T = 0$
for $i \in [1...n]$ **do**
    $M_{i0} = T + S(-, A_i)$
    $T = M_{i0}$
end for
$T = 0$
for $j \in [1...m]$ **do**
    $M_{0j} = T + S(-, B_j)$
    $T = M_{0j}$
end for
$M_{00} = 0$

After the 0 row and column are initialized the rest of M can be computed. The alignment score at M_{ij} is found by comparing and choosing the maximum of the scores that would happen if A_i was matched with B_j, or A_i was paired with a gap, or B_j was paired with a gap. Pseudo-code is presented in Algorithm 2. Once M is computed the entry at M_{nm} contains the optimal alignment score for A and B.

Algorithm 2 Needleman-Wunsch algorithm

for $i \in [1...n]$ **do**
    for $j \in [1...m]$ **do**
        $match = M_{i-1,j-1} + S(A_i, B_j)$
        $gapA = M_{i-1,j} + S(-, A_i)$
        $gapB = M_{i,j-1} + S(-, B_j)$
        $M_{i,j} = max(match, gapA, gapB)$
    end for
end for

Notice that Algorithm 2 computes M but does not compute the aligned strings A' and B'. Fortunately, the aligned strings can easily be reconstructed by backtracking through M. This is done by starting at $M_{n,m}$ and checking to see if the value there corresponds to $match$, $gapA$, or $gapB$ being chosen in the corresponding iteration of Algorithm 2. If $match$ was chosen A_n and B_m are aligned and we can move to $M_{n-1,m-1}$. If $gapA$ was chosen A_n is aligned with a gap and we move to $M_{n-1,m}$. If $gapB$ was chosen B_m is aligned with a gap and we move to $M_{n,m-1}$. This process is repeated until M_{00} is reached.

3.2.2 Similarity Metric

We use the alignment score as part of a similarity metric, as proposed in [50]. The authors of that work propose a method for computing similarity between two sequences representing web usage patterns for two users of an online system. They let characters represent different web pages. A web usage session is then a string of characters showing the order that users visited web pages in. The authors use sequence alignment to compute similarity between strings (letting them quantify similarity between sessions).

The actual similarity score used combines scores for the actual alignment and the significance of the alignment. The alignment score is computed as a ratio of the alignment score taken from M and the alignment score when just considering characters that were correctly matched. Duraiswamy et al. use the max value in M instead of $M_{0,0}$, but we feel both methods are worth considering (since $M_{0,0}$ gives the actual global alignment score).

$$dis(A, B) = M_{0,0}$$

$$S'(a, b) = \begin{cases} S(a, b) & \text{if } a = b \\ 0 & \text{if } a \neq b \end{cases}$$

$$dis_{correct}(A, B) = \sum_{i=0}^{n} S'(A_i, B_i)$$

$$Sim_{align}(A, B) = dis(A, B)/dis_{correct}(A, B)$$

The second part of the score, which scores the significance of the alignment, is a ratio of the number of characters correctly matched to the total length of the alignment.

$$cor(a, b) = \begin{cases} 1 & \text{if } a = b \\ 0 & \text{if } a \neq b \end{cases}$$

$$Num\_Correct(A, B) = \sum_{i=0}^{n} cor(A_i, B_i)$$

$$Sim_{Significance}(A, B) = Num\_Correct(A, B)/n$$

Then we can put together the two scores to get a similarity metric. The similarity metric can have values between -1 and 1, with higher values denoting a higher similarity between two sequences.

$$Sim(A, B) = Sim_{align}(A, B) * Sim_{Significance}(A, B)$$

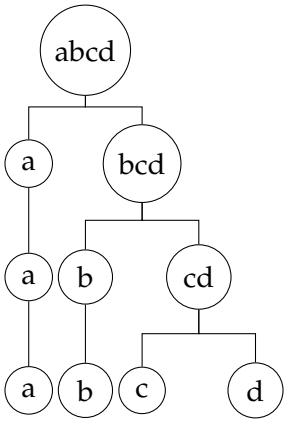
3.3 Clustering

Machine learning can be categorized based on two dimensions: one being *supervised* or *unsupervised*, the other *classification* or *regression*. Supervised machine learning methods use labeled data i.e. a *training* data set is available that has explicit target values for each given example (data-point). Unsupervised methods work without pre-labeled data. Unsupervised techniques try to discover structure or meaning in a data-set. On the other dimension, classification tasks are a class of problems where examples need to be mapped to elements of a finite set of values. The target values are referred to as *classes*. Regression is when examples are mapped to a real number. Regression and classification are closely related, as they both concern making predictions about examples.

For this section of the thesis we are most concerned with unsupervised classification. How does one make sense of an unlabeled data-set? How can the different examples be separated and categorized as to add meaning to the data-set? The most common approach to unsupervised classification is called *clustering* [51]. Clustering is the process of separating data-points into groups known as *clusters*. A particular way of assigning examples to clusters can also be known as a *clustering*.

There are a few distinctions that can be made regarding clusterings. One is the notion of *hierarchical* clustering. A standard clustering is *partitional* or non-hierarchical, meaning that each example is assigned to one cluster and clusters are simply sets of examples. A hierarchical clustering is one that allows *sub-clusters*. Clusters are sets of clusters, except for clusters of a single element, which will contain an example (these are also known as *leaves*). Hierarchical clusterings are easily visualized as trees, as shown in Figure 3.1. Note that a hierarchical clustering implicitly gives a sequence of partitional clusterings, because at each height of the tree there is a partitional clustering. The partitional clustering is obtained by pruning all nodes with a height greater than the desired height and selecting the leaf nodes. For example, in Figure 3.1 the partitional clustering $\{a\}$, $\{b\}$, $\{c, d\}$ can be obtained by pruning the tree at height 3. As will be explained shortly, we used hierarchical clustering techniques, with the end goal of obtaining partitional clusterings.

Figure 3.1: Hierarchical Clustering



Clusterings can be *exclusive* or *overlapping*. Exclusive clusterings assign examples to single clusters. Overlapping or non-exclusive clusterings allow examples to belong to multiple clusters. Overlapping clustering is appropriate for situations where multiple categories can be assigned to a single example. For example, an animal can live in water and on land. For this work we chose to use exclusive clusterings because we want to be able to describe build orders with single strategies (mostly for the sake of constructing game matrices). Also related is the concept of *fuzzy clustering* [52] where each example has a vector of real numbers in $[0, 1]$ where each dimension in the vector represents the degree of the example's assignment to a corresponding cluster. Fuzzy clustering is used in situations where even human labeling of data tends to cause ambiguities between what examples should be in what clusters.

A *complete* clustering is when every example in a data-set is assigned to a clustering. Alternatively, a *partial* clustering is when some examples may not be assigned to a cluster. Partial clusterings are used primarily to deal with data-sets that contain outliers. Although the techniques we look at here typically generate complete clusterings, we will also present a simple method for computing a partial clustering from a complete clustering. A partial clustering is appropriate for clustering strategies, because it is possible that not every game replay in a given data-set will have a coherent high level strategy associated with it [40].

Since a data-set can be clustered in different ways, a common issue in data-mining is deciding which clustering is the best. This could mean better for a specific

task, or better at providing insight into understanding the data-set. In general, cluster evaluation is a difficult problem, although there are properties of clusters that are desirable, depending on the context. Different properties might be desired for different data-sets. For our work, we refer to the properties of *well-separatedness* and *conceptual coherence*. A clustering is well-separated if examples in a cluster are closer (more similar) to the other examples in that cluster than to examples in any other cluster [53]. The well-separated property is valued when "natural" clusters are believed to exist in the data-set. We use well-separatedness in this work because there have been shown to be naturally distinct strategies in RTS games before [25]. Conceptual coherence refers to the semantics of the clusters. Clusters should represent some abstract idea, and the examples in the cluster should reflect that idea. For clustering build-orders, we want clusters to represent strategies and for the build-orders themselves to reflect the strategy of the cluster in some way.

3.3.1 Agglomerative Hierarchical Clustering

To cluster build-orders we chose to use a well known hierarchical clustering technique called *agglomerative hierarchical clustering*. Agglomerative hierarchical clustering was chosen because of relative simplicity and because the input to the algorithm is just a similarity matrix [54]. There are two main types of hierarchical clustering techniques: *agglomerative* and *divisive*. Agglomerative techniques begin with each example in its own cluster. Each level of the hierarchy is then created by merging clusters together. Divisive techniques work in the opposite direction. They start with all of the examples in a single cluster, and create the hierarchy by splitting up clusters according to some policy. Agglomerative techniques are more common and tend to be the type that data mining software packages implement. We chose to use an agglomerative technique for the work presented here because of their widespread use and simplicity.

The general agglomerative hierarchical clustering algorithm is fairly simple. The algorithm takes a similarity matrix between examples as input. Also needed is some notion of cluster *proximity*. Proximity is a measure of closeness between clusters (like similarity is to examples). Proximity is usually computed as a function of the similarity scores of examples e.g. a common proximity measure is the maximum similarity score between the examples of two clusters. The first step of the algorithm is to re-compute the similarity matrix in terms of cluster proximity (de-

pending on the chosen proximity measure this may not be necessary). So now the rows and columns of the matrix represent clusters. Cluster merging is then done iteratively until all the examples are in a single cluster. Each iteration, the clusters with the highest proximity are chosen and merged together. The proximity matrix is then updated to reflect the merge. Algorithm 3 formalizes this method.

Algorithm 3 Agglomerative Hierarchical Clustering

```
Input: Similarity matrix S
Compute proximity matrix P from S
while |P| > 1 do
    Merge clusters i and j where P_ij is maximized
    Update P
end while
```

The most common proximity measures are max, min, and average. Max uses the two cross-cluster examples closest to each other as a measure for the proximity of the two whole clusters i.e. it selects the maximum similarity that includes a member examples from each cluster and uses that as the cluster proximity. Min uses the two cross-cluster examples which are farthest away from each other (so it minimizes similarity). Average takes the average similarity among all pairs of examples between the two clusters, and uses that value as the proximity between the two clusters. We decided to use the average proximity measure because it takes into account all the examples of each cluster, and we desire that clusters have the well-separated property. Formally, for two clusters C_1 and C_2 and a similarity matrix S, the proximity measures are:

$$Max(C_1, C_2) = max(\{S_{ij} | i \in C_1, j \in C_2\})$$

$$Min(C_1, C_2) = min(\{S_{ij} | i \in C_1, j \in C_2\})$$

$$Average(C_1, C_2) = \frac{\sum_{i \in C_1} \sum_{j \in C_2} S_{ij}}{|C_1| * |C_2|}$$

Another linkage policy in use is the Ward linkage policy [55]. Ward's method is distinct from the other methods because it attempts to incorporate a global objective into the merge step. The clustering itself still cannot be viewed as having a global objective function, but at the merge steps the policy chooses to do the merge

that minimizes the squared error. Ward's method can be implemented as a recursive formula, allowing it to just use the similarity matrix as a basis for the proximity matrix (which can be updated over each iteration). Let C_1' and C_1'' be the two clusters which were merged to make C_1. Let $n_1' = |C_1'|$, $n_1'' = |C_1''|$ and $n_2 = |C_2|$. We can then look at proximity in terms of:

$$\begin{aligned} Ward(C_1, C_2) = & \frac{n_1' + n_2}{n_1' + n_1'' + n_2} Ward(C_1', C_2) + \frac{n_1'' + n_2}{n_1' + n_1'' + n_2} Ward(C_1'', C_2) \\ & - \frac{n_2}{n_1' + n_1'' + n_2} Ward(C_1', C_1'') \end{aligned}$$

The Ward linkage ends up behaving similar to the $Average$ linkage policy. All the linkage policies discussed here are specific parameterizations of the more general Lance-Williams formula [56]. The Lance-Williams is simply a generalized form of proximity measure that includes Max,Min,$Average$, and $Ward$.

3.4 Applied to StarCraft

We have now introduced a scheme for clustering sequences. Sequence alignment is used to provide similarity between sequences. Agglomerative hierarchical clustering is used to cluster the resulting similarity matrix. This section will give details about how the scheme can be applied to data-mining replays of a real-time strategy game, along with some experimental results. As explained earlier, we use the game StarCraft for experimental purposes. StarCraft has had a large user community, has freely available replays online, and is the main real-time strategy game that AI systems compete in.

3.4.1 Data

The work presented in this section of the thesis uses the parsed data provided by Synnaeve et al. [31]. We looked at the datasets for the Protoss-versus-Protoss (PvP) and Protoss-versus-Terran (PvT) match-ups. Synnaeve's parsed data-set contains various kinds of information about each match, kept in one of three text files. In one of the text files, events signifying the creation of a new unit are recorded along with timestamps. This information seemed recorded mostly properly at least, and can be parsed easily to get the build-order sequences for each game. We mapped

each unit to a unique character, so build-order sequences are simply strings that represent the order units were built in.

Synnaeve's dataset includes most of the amateur replays freely available at the time that the dataset was released (which was mid 2012). The dataset contains many matches between proficient players, but since the games are of an amateur nature not all of the matches contain high-level play. The majority of the dataset contains good matches though, so we regard the outliers as noise. We focus on replays that use the Protoss faction for a few reasons. The AI system being developed at the University of Alberta (named UAlbertaBot) plays primarily with the Protoss faction, so any findings would be most useful if they dealt with Protoss. The majority of the AI systems being developed play with Protoss, so related works tend to deal with Protoss [4]. Finally, this a preliminary work (the problem has not been dealt with for StarCraft, as far as we know), and Protoss has relatively simple mechanics. The PvP dataset is smaller and symmetric and was used in testing, and the PvT dataset is larger and non-symmetric. We chose Terran over Zerg simply because of the author's lack of familiarity with the Zerg faction.

We follow Jeff Long's scheme of just using units for build-orders. That is, we exclude buildings and workers from the build-orders. We exclude buildings because there is a large correlation between building existence and the types of units being made, and units are much more indicative of the actual strategy being carried out (so including buildings would not provide significant information gain and would increase computation time). Leaving out workers could be seen as a more controversial decision. Workers are a part of every build order, so their presence is not indicative of strategy. A counter-argument is that the order workers and other units/structures are built are crucial to identifying different strategies, especially in the early game. We argue that such subtleties are most important in the early game and we are concerned with strategic abstractions that encompass the whole game. For a study that looked only at openings, including workers would probably be useful. Also, when worker creation order is important, it is usually in terms of the order workers and buildings are created in (e.g. probes and pylons for the Protoss faction). Since buildings are being excluded, workers may not be particular illuminating.

Recall that each unit type is mapped to a unique character in order to encode build-orders as sequences. The set of characters is our alphabet for the sequence

alignment. Since we explore both Protoss and Terran build-order clusterings, there are alphabets for both factions. The Protoss alphabet contains 15 characters and the Terran alphabet contains 12 characters. Table 3.1 shows the characters assignments for both alphabets. The Protoss alphabet is on the left and the Terran alphabet is on the right. Note that upper and lower case characters are different characters.

Observer	B	Marine	A
Dragoon	D	Ghost	B
Zealot	E	Vulture	C
Archon	I	Goliath	E
Reaver	J	Siege Tank	F
High Templar	P	Wraith	G
Arbiter	T	Science Vessel	H
Carrier	W	Battlecruiser	I
Shuttle	Y	Firebat	K
Scout	Z	Medic	L
Pylon	e	Valkyrie	M
Nexus	g	Dropship	N
Corsair	m		
Protoss Dark Templar	n		
Protoss Dark Archon	A		

Table 3.1: Alphabet

3.4.2 Unit Similarity

Recall that the Needleman-Wunsch sequence alignment algorithm requires a similarity function (also called a cost function) that rewards matches and penalizes mis-matches between characters. We could use a naive, domain-independent cost function like the edit distance, but we feel it is more interesting to propose a domain particular cost function that adds semantics to the characters themselves. Since our domain is StarCraft and the characters in the sequences are units, we propose to use a cost function that operates on in-game attributes of the units themselves. Let ρ be a cost function on two characters a and b which represent units. The general form of ρ is

$$\rho(a,b) = \begin{cases} k * c(a) & \text{if } a == b \\ |c(a) - c(b)| - \phi(a,b) & \text{otherwise} \end{cases}$$

where c is a function of a single character that assigns a singular value to that character, k is a constant, and ϕ is a function of two characters that works as a

possible additional mis-match penalty. ϕ is there to provide an additional sense of how different two characters are. ϕ should also use attributes of the units a and b represent, although the chosen attributes do not need to be the same as what c uses. ρ is a general form for a cost function, and the choices for k, c, and ϕ decide the particular cost function.

ρ is much like the cost function used by Jeff Long [40], and is in-fact a generalization of the custom cost function presented in that work. That work was done on a different RTS game called *WarCraft III*, where units have a *food* attribute that describes the economic cost of the unit. Food value for a unit a is used as the value of $c(a)$ and they use 16 as the value of k. The value of 16 for k seems to be the result of trial-and-error, although the exact value may not be that important (k should just be relatively small or large compared to c depending on the level of importance one wants to place on correct matches).

We decided to use the in-game of concept of *supply* as the analogous measure to food for StarCraft. Supply is a mechanic in StarCraft that limits how many units a player can have at one time. Each player starts with a supply cap, and they can build certain buildings to increase the supply cap (this works slightly different depending on the faction choice). Every unit has a supply value associated with it. A player's total supply cannot exceed their supply cap. Since different units have different supply values based on their perceived importance by StarCraft's designers, supply seemed like a good measure of value. For our implementation of ρ, we use a units supply as the value for c. We decided to use 16 for the value of k, because supply values are not significantly different in magnitude from food values, and we desire to place high importance on correct unit matches.

ϕ is a function that should not penalize mis-matches when the units are relatively similar, and should punish mis-matches where the units are fairly different. In Jeff Long's implementation of ρ, ϕ is zero when a and b are produced from the same building and a non-zero constant when a and b are produced from different buildings (the value of the constant was not provided). Inspired by Jeff Long's penalty, we propose the use of a varying penalty depending on how categorically similar a and b are. For deciding unit similarity, we design a sort of unit *ontology* based on expert knowledge. Units can be categorized based on their use and attributes. We propose to use an ontology (which is a hierarchical categorization) which is hand crafted based on the units themselves. If units lie in the same im-

mediate categorization then ϕ is small. If the units differ by their immediate categorization but fall in the same parent category of their immediate categorization, then ϕ is slightly larger, and so on.

We used knowledge of the unit types to design the ontology. Ontologies were designed for Protoss and Terran, with Protoss presented here. The most severe categorical distinctions for Protoss can be seen in Figure 3.2. At the highest level of the ontology we separate units into three categories. *Drop* refers to drop ships (used to transport other units). *Recon* refers to information gather units, which for Protoss is the Observer. *Military* refers to combat units, which are the rest of the units. If units differ by the highest layer of the ontology, ϕ is an integer value of 4 (which is our most severe mis-match penalty). Military is then divided into *flying* and *ground* depending the units mobility. If units differ at this level then ϕ is 3. The divisions in ground represent the lowest categorical differences and can be seen in Figure 3.3. Ground is divided based on the unit's primary mode of attack. *Ranged* is for units that attack from afar, *Melee* units attack close up, and *Spellcaster* is for units that cast spells (which have various purposes). Units that differ within either ground or flying get a penalty of 2. Units that are in the same bottom category get a penalty of 1, which is the smallest value we decided to use for ϕ.

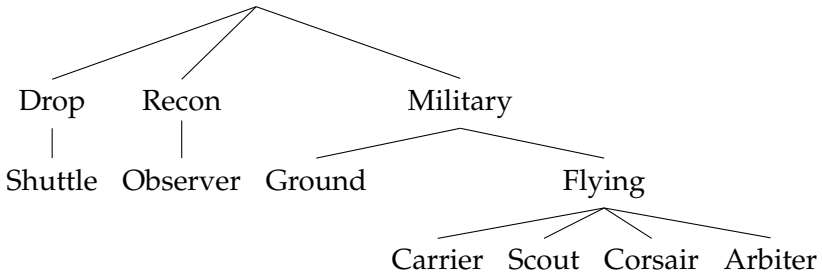


Figure 3.2: Top layers of the Protoss Ontology

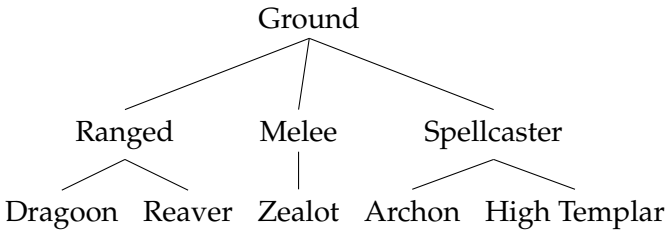


Figure 3.3: Bottom layers of the Protoss Ontology

We used the custom cost function presented here in clustering both the Protoss-versus-Protoss (PvP) and Protoss-versus-Terran (PvT) datasets. ϕ for the Terran cost function uses an analogous ontology for assigning penalties. The only major differences is that there is no recon category and instead of spellcaster we just used single unit categories for the units medic and ghost (which are unique units).

To give additional insight into the nature of the alignments being done in this work, we present an example of a real build-order sequence alignment. The following example is using Protoss build-orders from the Protoss-versus-Terran dataset and uses the Protoss alphabet found in Table 3.1. Let there be two independant build orders:

EEEEEDDDDnnDDDDDD

EEEEDDDDDDEEDDYEJDDDDJDDDDY

Using the custom Cost function described previously, the alignment given by the Needleman-Wunsch algorithm is:

_ EEE _____ EEDD __ DDnnDD _ DDDD _

EEEEDDDDDDEEDDYEJDD__ DD J DDDDY

Before evaluation of the clustering process is explored, we feel it important to give some insight into how effective the sequence alignment procedure is at producing meaningful alignments. Let A and B be two build orders being aligned and let n and m be their respective lengths. The minimum length of the aligned strings is given by $best = max(n, m)$ and the maximum length is $worst = n + m$. Alignment lengths that are closer to $best$ involve fewer gaps and suggest that the alignment contains information about the sequence. Alignments that are closer to $worst$ involve a large number of gaps and show that the two sequences aren't being aligned so much as mis-aligned. To investigate the effectiveness of our scheme in providing meaningful alignments we suggest the following metric:

$$\frac{actual - best}{worst - best}$$

where $actual$ is the length of the aligned strings. When the metric is 0 the alignment represents a best case. When the metric is 1 the alignment represents a worst case. We chose subsets of the Protoss-versus-Terran dataset to examine the effectiveness of the alignments. Figure 3.4 shows the alignment metric on alignments between short build-orders. In particular, figure 3.4 involves all alignments between build-orders with a length less than 50 units. It can be seen that for short build-orders our alignment scheme produces reasonable alignments. Figure 3.5 shows the alignment metric between longer build-orders. For figure 3.5 we analyze alignments between build-orders between 200 and 250 units in length. Alignments on longer build-orders are still reasonable in terms of the alignment metric, but the overall trend suggests that as build-orders grow in length the effectiveness of our technique diminishes.

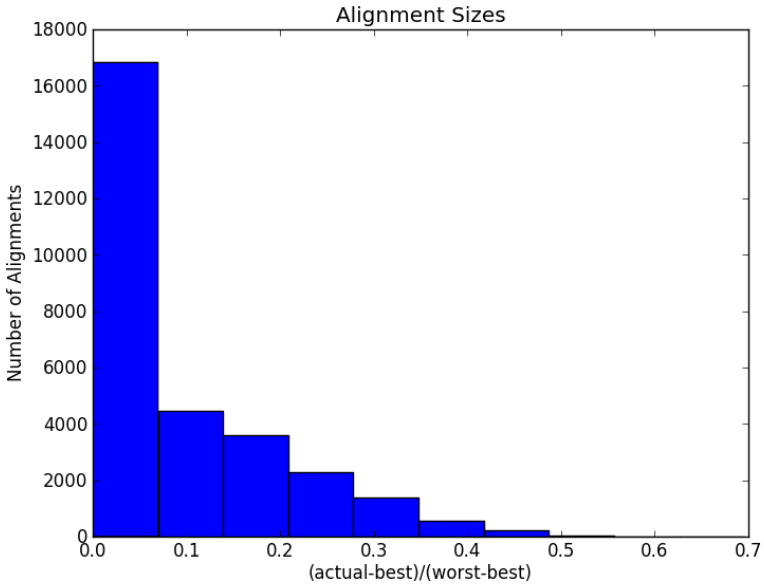


Figure 3.4: Alignments between build-orders from the PvT dataset less than 50 units in length

3.4.3 Cluster Evaluation

Two issues became apparent when we started experimentation: How should a clustering be evaluated (i.e. what makes a clustering good?) and what level of the hierarchical clustering should be chosen to create a partitional clustering. These two concerns are not unrelated. If we have a technique for evaluating a partitional

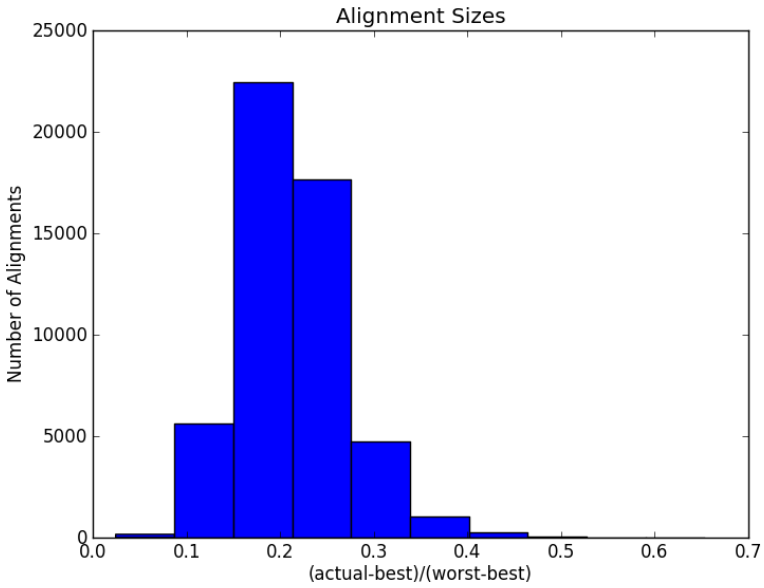


Figure 3.5: Alignments between build-orders from the PvT dataset between 200 and 250 units in length

clustering, then that technique can be used to select the level of the hierarchy to use as a partitional clustering. The idea would be simply to iterate over the levels of a hierarchical clustering, selecting the current leaves as clusters in a partitional clustering and evaluating. Levels that maximize the evaluation (in either a global or a local sense) make good candidates for clusterings. Evaluation metrics will be explored shortly.

We also consider our end goal when deciding if a clustering is good or not. Since the purpose of the work in this chapter is to build payoff matrices, a good clustering is a clustering that can be used to build a payoff matrix that offers enlightening information. If one can learn something useful from a payoff matrix, then the clusterings which were used to build the matrix can be considered good.

Since we are using agglomerative hierarchical clustering as the primary clustering technique, it is also desirable to have a metric for evaluating different hierarchical clusterings. From a set of different hierarchical clusterings, a single hierarchical clustering that maximizes such a metric could be chosen, and that clustering could be examined to select a partitional clustering. Fortunately, a well-established metric for selecting hierarchical clusterings exists.

The *CoPhenetic Correlation Coefficient* can be used to evaluate hierarchical clus-

terings [57]. Recall that the agglomerative hierarchical clustering algorithm uses a proximity matrix P. Also recall that at the start of the agglomerative process all the examples are in their own clusters, and at the end of the process all the examples are in the same cluster. So for two examples x and y, there will be an iteration i when x and y are first in the same cluster. For iterations $< i$, x and y will be in different clusters and for iterations $\geq i$, x and y will be in the same cluster. For every pair of examples x and y, i exists. At the start of iteration i, $x \in C_j$ and $y \in C_k$ where C_x and C_y are clusters and $C_x \neq C_y$. Let j and k be the indices into P that represent C_j and C_k respectively. Then for every $x \in C_j$ and $y \in C_k$, $P_{j,k}$ is the proximity that x and y were first chosen to be in the same cluster. Note that because of the need for a proximity measure, this value is not the same as the original similarity between x and y, S_{xy}. $P_{j,k}$ is known as the *cophenetic distance* between x and y. Also note that when two clusters C_j and C_k are first merged, the cophenetic distance is the same for every pair x and y where $x \in C_j$ and $y \in C_k$. Also note that cophenetic distance depends on a hierarchical clustering and not just the set of examples. So cophenetic distances can be different for different clusterings of the same set of examples.

For every pair of examples x and y in the dataset, a cophenetic distance can be computed. This allows the creation of a cophenetic distance matrix P'. Every example is represented by P', so it is the same size and shape as S. The values in P' can be the same as the values in S but most of them will be different. The cophenetic correlation coefficient (CPCC) is then the correlation between S and P'. Basically it says to what degree a hierarchical clustering reflects the distances that exists between examples. Higher CPCC values show a closer fit between clustering and data, and lower values show a worse fit. The CPCC is most useful for trying out different linkage policies and selecting the policy with the largest CPCC. Remember that the linkage policies presented in this thesis are Min,Max,$Average$, and $Ward$.

To show how the CPCC can be used to select a linkage policy, we have computed cophenetic distance matrices and their corresponding CPCC for clusterings using different linkages on two of our datasets. The numbers presented here are from the units only versions of the Protoss-versus-Protoss (PvP) and Protoss-versus-Terran (PvT) data-sets, excluding workers and using the custom score matrix during sequence alignment. The PvP results can be found in Table 3.2. Since the PvT

dataset involves two separate clusterings (one for each faction), the results for the PvT dataset are reported separately for each race, as found in Table 3.3. In all cases, the $Average$ linkage policy yields the highest CPCC value.

Linkage Policy	CPCC
Min	0.62337
Max	0.21094
$Average$	0.76905
$Ward$	0.56441

Table 3.2: CPCC values for PvP data using different linkage policies

Linkage Policy	Protoss CPCC	Terran CPCC
Min	0.68256	0.77136
Max	0.18612	0.16551
$Average$	0.83518	0.85562
$Ward$	0.61552	0.54474

Table 3.3: CPCC values for PvT data using different linkage policies

At the beginning of this subsection we mentioned how a metric for evaluating partitional clusterings could be used to select the level of a hierarchical clustering used as a partitional clustering. Here we will present a metric for cluster evaluation and show the level selection process on real data. Recall that the desired metric rewards well-separatedness in a clustering. We use the formal concepts of *cohesion* and *separation* [51]. The well-separated quality requires that examples in the same cluster are similar to each other and are different from examples in other clusters. Cohesion is a measure of how similar a cluster is to itself. We define it here for a cluster C simply as a sum of the similarities between member examples, using the similarity matrix S:

$$Cohesion(C) = \sum_{i \in C} \sum_{j \in C} S_{i,j}$$

Cohesion is a function of a single cluster. Separation is a function of two clusters, C and C'. Separation quantifies how different two clusters are from one another:

$$Sep(C, C') = \sum_{i \in C} \sum_{j \in C'} S_{i,j}$$

Sep gives the separation between two clusters. Separation can also be viewed as a measure of a single cluster's separation from all other clusters. Let κ be the clustering currently being evaluated, of which C is a member:

$$Separation(C) = \sum_{\substack{C' \in \kappa \\ C' \neq C}} Sep(C, C')$$

Above Cohesion and Separation are defined for clusters. We can also define a combined metric that measures Cohesion and Separation for a clustering κ. The metric Sep_and_Co captures the idea of Well-Separatedness for a clustering. Notice that well-separatedness requires $Cohesion$ to be high and $Separation$ to be low. So the smaller Sep_and_Co is, the more well-separated κ is.

$$Sep\_and\_Co(\kappa) = \sum_{C \in \kappa} \frac{Separation(C)}{Cohesion(C)}$$

We will now report how we used the metric to investigate the Protoss-versus-Protoss (PvP) dataset. This is using the hierarchical clustering obtained using build-orders with units only, our custom scoring matrix for the sequence alignment and the $Average$ linkage policy. Figure 3.6 shows the Sep_and_Co score for partitional clusterings taken at each level of the hierarchical clustering of the PvP dataset. The top (or last in terms of iterations of the agglomerative clustering method) level of the hierarchical clustering corresponds to a partitional clustering containing a single cluster. The next level corresponds to a partitional clustering with two clusters, and so on. So in the following diagrams, when we report the number of clusters in a clustering, that number is also referring to the level of the hierarchical clustering that the partitional clustering was taken from. The domain of the number of clusters is from 1 to n where n is the number of examples.

It is obvious from Figure 3.6 that the Sep_and_Co score on its own is not very enlightening. One problem is that $Separation$ is zero for clusterings of a single cluster. The other problem is that there is simply less cross-cluster comparisons involved in generating the $Separation$ part of the score for lower cluster counts. We attempt to rectify these issues by normalizing with the number of clusters involved in the clustering. We disregard clusterings of a single clustering, since that is not very enlightening, and point out that since identifying possibly interesting levels to take partitional clusterings from, local minima could be just as useful as global minima. Figure 3.7 shows the same data but normalized by the number of clusters.

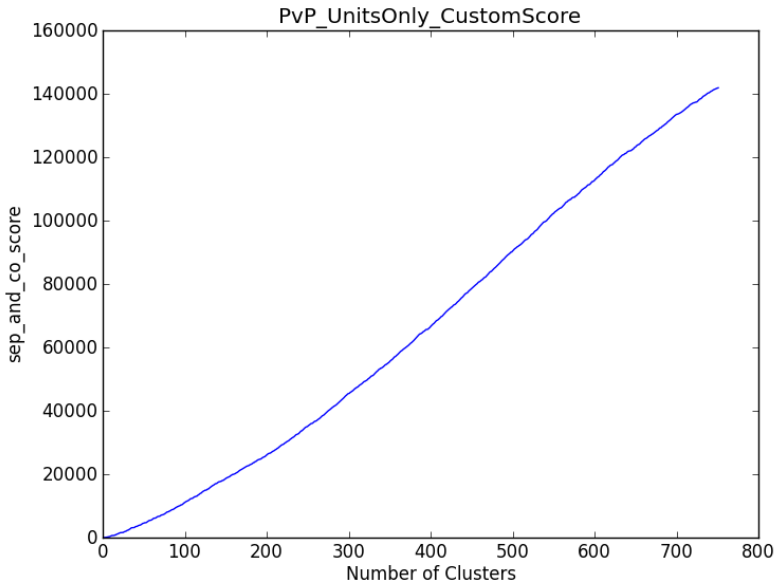


Figure 3.6: Sep_and_Co versus the number of clusters for the hierarchical clustering of the PvP dataset

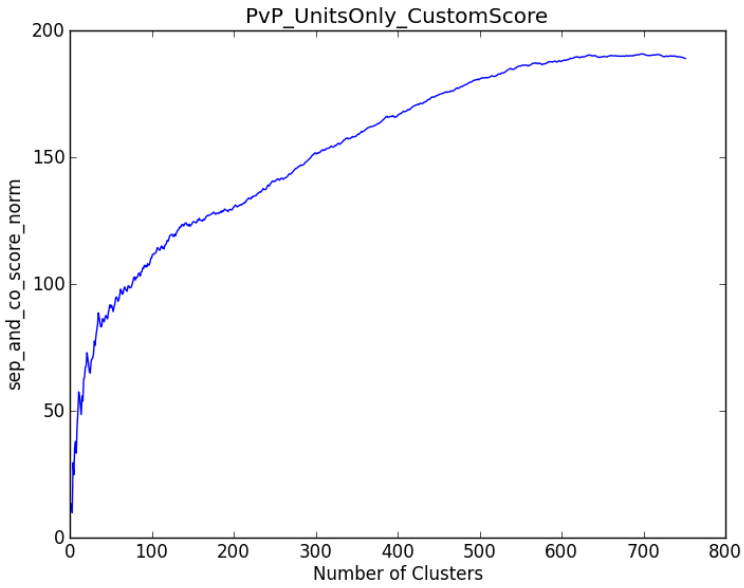


Figure 3.7: Sep_and_Co versus the number of clusters for the hierarchical clustering of the PvP dataset normalized by number of clusters

Figure 3.7 shows signs of local minima. Clusterings with a large number of clusters (large meaning $n > 50$) are not particularly useful, since one of the goals of this work is to provide steps in building human-readable payoff matrices. Figure 3.8 shows a close up view of levels 2 to 100. Local minima are labeled with their

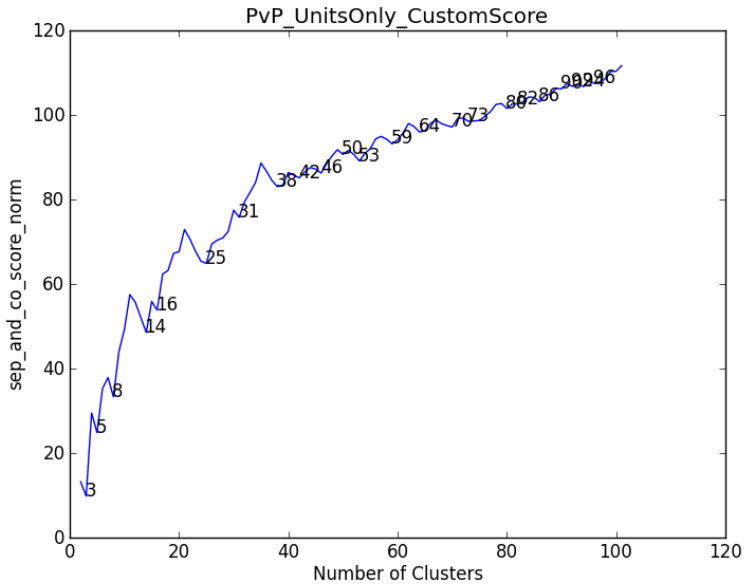


Figure 3.8: Sep_and_Co versus the number of clusters for the hierarchical clustering of the PvP dataset normalized by number of clusters on the domain of [2,100]

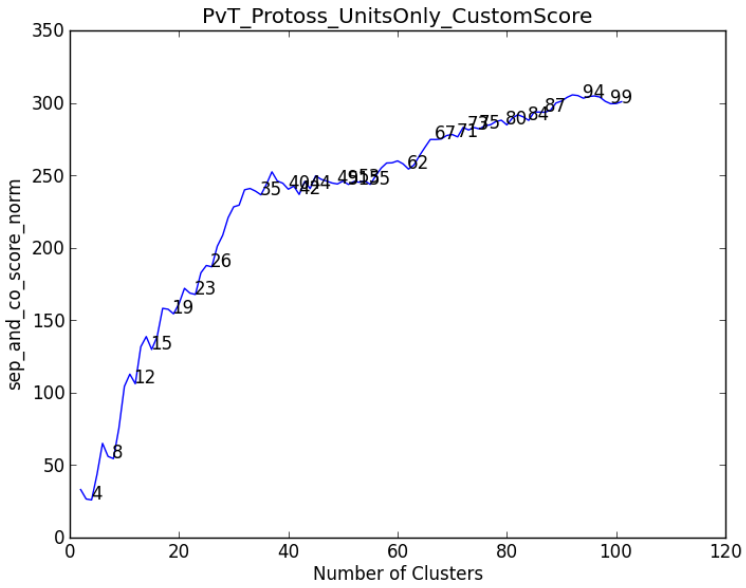


Figure 3.9: Sep_and_Co versus the number of clusters for the hierarchical clustering of the PvT dataset normalized by number of clusters on the domain of [2,100] just using Protoss players

respective number of clusters.

As shown by Figure 3.8, the clustering with 3 clusters globally minimizes Sep_and_Co if clusterings with a single cluster are disregarded. The clusterings of size 5 and 8

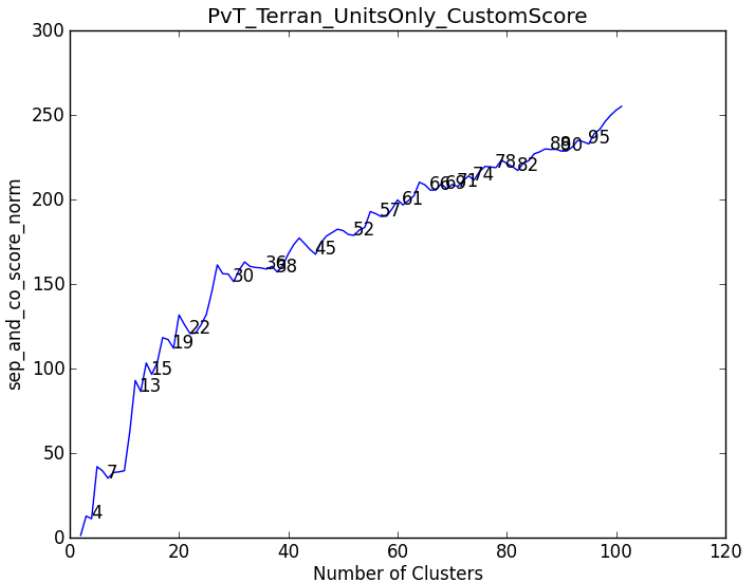


Figure 3.10: Sep_and_Co versus the number of clusters for the hierarchical clustering of the PvT dataset normalized by number of clusters on the domain of [2,100] just using Terran players

are also of interest, as they are local minima and relatively small in the number of clusters. One of the issues with using clusterings that contain many clusters is that in order for a payoff matrix based on the clustering to be well populated, a large dataset is required. Since our dataset (especially for the PvP dataset) is somewhat small, the clustering with 3 clusterings seems like the most appropriate choice.

We can do a similar process for level selection in clusterings done on the Protoss-versus-Terran (PvT) dataset. The figures presented here use the hierarchical clusterings of the PvT dataset using only units in the build-orders and the custom scoring function during sequence alignment. PvT is different from PvP because it is a non-symmetric match-up and it is significantly larger. Since PvT is non-symmetric it requires two clusterings: one for build-orders just using Protoss players and one for Terran players. For PvT we will just show the close-up and normalized versions of the Sep_and_Co plots, since they are what contain the interesting information. Figure 3.9 shows the Protoss clusterings and figure 3.10 shows the Terran clusterings

Figure 3.9, the Protoss plot, shows the global minima at the clustering with 4 clusters, with 3 being very close to the global minima. Overall, figure 3.9 is similar to figure 3.8. Figure 3.10, however, has some differences. We observe a local

minima 4, but the global minima is at 2. 2 clusters however is not as interesting, especially since one of the clusters has a much larger population than the other (and we would like our examples to be spread out between clusters).

3.4.4 Building Payoff Matrices

Now that clustering build orders has been discussed, it is important to be able use the discovered clusters for something. We propose using the clusterings to construct payoff matrices, which continues in the vein of Jeff Long's work. Although the clustering techniques presented here have some interest in and of themselves, mainly in terms of identifying natural groupings of build orders, forming payoff matrices moves the work from purely academic to having more practical applications. The usage Jeff Long recommends is for analyzing the balance of a game. A major limitation of Jeff Long's work is that at least a training set of games has to be hand labeled by experts. This is time consuming and introduces human error. It also does not allow novel strategies to emerge that humans might not have a high-level abstraction for. Using clustering provides researchers a method (or a set of methods) for automating balance detection for a game, provided that a sufficient amount of replays for the game are available. Jeff Long's thesis discusses the game theory behind the properties a payoff matrix must hold for a game to be considered balanced.

The other way such a payoff matrix can be used is to identify counter strategies. Work has been done on predicting opponent strategy [4] [27], showing that detection of opponent strategy is possible. If in a game, a player learns the opponent's strategy, it would be desirable to have a known counter-strategy to employ (i.e. a strategy that is known to be effective at defeating the opponent's strategy). If a payoff matrix for strategies exists then the problem is trivial; look up the opponent's strategy and find which player strategy has the highest probability of winning and employ that strategy. Depending on the game and on the data, these rock-paper-scissors patterns might not exist. Some RTS games might just be very well tuned, so that in terms of high-level strategies, strategies do not dominate other strategies. The other problem is that if the replays are from matches between proficient players, players may be smart enough to not employ losing strategies. In this case, rock-paper-scissors patterns would exist in the game, but do not show up in the data and would not be visible from a payoff matrix built from the data.

Building the matrix is also simple. The rows of the matrix represent the strategy clusters for the player found during the clustering. The columns represent strategy clusters for the opponent. The clusters contain build-orders, which came from game replays. So each cluster can be associated with a set of matches. Each match involves two players, and so can be connected to both a cluster on the rows and a cluster on the columns (note that for a symmetric match-up these might be the same cluster). The elements of the payoff matrix can then be populated in terms of the results of the matches. Let G be the payoff matrix being populated. The form suggested by Jeff Long is as follows:

$$G_{ij} = \frac{w_{ij} - l_{ij}}{t_{ij}}$$

where w_{ij} is the number of wins for cluster i (the row player) that can be found from games between cluster i (a row) and cluster j (a column). l_{ij} is the number of losses for cluster j and t_{ij} is the total number of matches between the two clusters (i.e. $t_{ij} = w_{ij} + l_{ij}$).

An entry G_{ij} is called a *payoff*. Payoffs can take values from -1.0 to 1.0. A payoff of 1.0 means that strategy cluster i wins against strategy cluster j all the time, and a payoff of -1.0 means that strategy cluster i loses against strategy cluster j all the time. A value of 0 means that i and j beat each other an equal number of times. Positive values mean that i is more likely to win and negative values mean that j is likely to win.

Because of the nature of the data, there are some cases where the payoff semantics are not intact. For starters, there can be matrix cells G_{ij} where no games can between i and j were observed. By the definition of G, the value of G_{ij} in that case is 0. This is different from the case where there were many observed matches and the number of wins and losses for i is equal. Similar is the case where there is only one observed match between i and j. In that case the payoff will be either 1.0 or -1.0, which looks like one strategy always wins against the other strategy, but that might not be true (especially if the one game used to populate G_{ij} is an outlier). When reporting payoff matrices built this way, we suggest reporting the value of t_{ij} along with G_{ij} so that semantic discrepancies are visible.

The other issue is the diagonal of the matrix in a symmetric match-up. Since the cells of the diagonal have the form G_{ii} and in the case of a symmetric match the rows and columns represent the same clustering, the diagonal represents games

from the same strategy cluster. Therefore the value at G_{ii} for a symmetric match-up will always be 0 (i.e. for every game in cluster i, one player will win, increasing w_{ii} by 1 and one player will lose, increasing l_{ii} by 1). So the diagonal of a symmetric match-up does not show particularly useful information using this scheme.

For the rest of this section, we present some examples of payoff matrices built using the Synnaeve data-set, along with a discussion of the information encoded by them. We first worked using the Protoss-versus-Protoss match-up, which is a symmetric match-up. We used 376 games for this experiment (we just used games that had clearly labeled winners and did not use any pre-processing/filtering). Since the match-up is symmetric, that gives 752 players. We used the clustering scheme shown in section 3.4.3 (i.e. the custom cost function was used for computing similarities and the average linkage policy was used for the agglomerative hierarchical clustering). We decided to use the partitional clustering with 3 clusters because of the results shown in figure 3.8 and explained in section 3.4.3.

	1	2	3
1	0 (0)	-1.0 (1)	0 (0)
2	1.0 (1)	0.0 (640)	0.2 (25)
3	0 (0)	-0.2 (25)	0 (60)

Table 3.4: Payoff matrix built from PvP data with 3 clusters

The payoff matrix for the PvP data can be found in table 3.4. The numbers along the rows and columns (outside the matrix) are simply labels. The labels along the rows correspond to the same clusters along the columns. The cells between the lines represent the cells of G. The first number listed is the value of the payoff. The second number (the one in brackets) is the value for t_{ij} (i.e. the number of examples used to populate the cell). Note that the value listed for t_{ij} is in terms of players (there are two players per match).

The most interesting part of table 3.4 is the cells $G_{3,2}$ and $G_{2,3}$. Over 25 matches, cluster 2 does defeat cluster 3 more often. Cells $G_{2,1}$ and $G_{1,2}$ are not particular enlightening because they represent singular examples (and along with the cells with zero examples show that cluster zero is simply lacking examples). We would like a matrix that shows several clear patterns like the one shown by cells $G_{3,2}$ and $G_{2,3}$. The cells along the diagonal offer little insight (as discussed before), and unfortunately the vast majority of examples in this data-set fall along the diagonal.

So the two main problems with the PvP matrix are the small number of examples in cluster 1 and the fact that the diagonal uses up most of the examples.

We propose working around the problems caused by a symmetric match-up by simply moving to a non-symmetric match-up. Next we will show the results found from constructing payoff matrices using the Protoss-versus-Terran data-set, which is significantly larger and non-symmetric. For the following experiment, we used 2089 games from the PvT match-up. This gives 2089 Protoss players and 2089 Terran players. As discussed previously in the chapter, a non-symmetric match-up requires two separate clusterings (one for each faction). The clusterings were both done using the custom cost functions and an average linkage policy. Also, because of the results shown in figures 3.9 and 3.10 and explained in section 3.4.3, we decided to use clusterings of size 4 for both factions.

	1	2	3	4
1	0 (2)	0 (0)	1.0 (2)	0 (0)
2	1.0 (1)	-1.0 (1)	0 (0)	0 (0)
3	-1.0 (1)	0 (0)	-0.09 (33)	0.07 (15)
4	0.15 (1858)	0.2 (10)	0.32 (60)	0.5 (4)

Table 3.5: Payoff matrix built from PvT data with 4 clusters

The payoff matrix built from the PvT data can be found in Table 3.5. The format is the same as for the PvP payoff matrix, although since this is for a non-symmetric match-up the row and column labels do not refer to the same clusters. For Table 3.5 the rows correspond to Protoss clusters and the columns refer to Terran clusters. This payoff matrix does not have the problems with the diagonal that the PvP matrix has. It also has a quite a few more cells that have potentially useful information (i.e. there are more non-zero cells). However, there still are many cells where t_{ij} is 0 or 1. Part of the problem is that the clusters are of vastly different sizes. Some clusters have the majority of the examples (like cluster 4 for the Protoss), while some clusters are still very small (like cluster 2 for the Protoss).

One idea is that the clusters with few examples do not represent coherent strategies, and act more like outliers or noise. Jeff Long found that when humans would label strategies in WarCraft III, some games would not fit a particular label (he made a special label called *none* for these examples). It might be beneficial then to select partitional clusterings such that small clusters are not included and only

larger clusters are kept.

We implemented a scheme for selecting a partitional clustering from a hierarchical clustering in such a way that would not include small clusterings. Let C be the topmost cluster of the hierarchical clustering, let P be a set of clusters that we are keeping for the partitional clustering (this is initialized to the empty set), and let T be a threshold size that determines how large a cluster needs to be to be kept. First we split C into two clusters C_L and C_R according to the hierarchical clustering (i.e. C_L and C_R are C's left and right children in the hierarchy). The larger of C_L and C_R is assigned to be the new value of C. If the smaller is larger than or equal to T it is added to P, otherwise it is discarded. This continues until P contains the desired number of clusters.

	1	2	3	4
1	0.07 (15)	-0.09 (33)	0 (0)	-1.0 (1)
2	0 (0)	1.0 (2)	-1.0 (1)	0.33 (3)
3	0.5 (4)	0.30 (158)	0.11 (9)	-0.03 (203)
4	0 (0)	1.0 (4)	1.0 (1)	0.17 (1655)

Table 3.6: Payoff matrix built from PvT data with 4 clusters using alternate cluster selection method

We used this scheme to select clusters to build a second payoff matrix for the PvT dataset with. Using a T of 5 we built a payoff matrix which can be seen in Table 3.6. This version of G for PvT has one less zero cell. There is still a single cell which has the vast majority of examples ($G_{4,4}$ in Table 3.6), but it is significantly smaller than in Table 3.5 (1655 examples from 1858 examples).

The cluster labels in the payoff matrices are just labels (they lack semantics). In order to get real meaning from the payoff matrices it is crucial to examine the cluster member elements. Quantitative methods and automated procedures for extracting meaning from clusters is left as future work. Here we will give a brief qualitative overview of the cluster contents, starting with the Protoss clusters (the row labels in Table 3.6). Cluster 1 consists mostly of smaller (in length) build-orders, with Zealots and Dragoons being the dominate units. The build-orders mostly embody rushing strategies. Cluster 2 is the smallest cluster, and scouts, shuttles, reavers, and carriers are all common in cluster 2. In particular, there are a few clusters that have shuttles and reavers built close (in the sequence) to each other. This suggests that these build-orders were used to implement the reaver-

drop strategy (which is a well known tactic that tries to drop reavers in the enemy base before the opponent is equipped to deal with them). Cluster 3 has mostly mid-length build-orders and is fairly Dragoon heavy. Qualitatively summarizing cluster 4 is particularly difficult because it is very large. The only clear unifying characteristic is that all the build-orders in cluster 4 are quite long. This shows that future work may benefit from breaking up large clusters (at least for the purposes of understanding cluster meaning). Moving to the Terran clusters (the column labels in Table 3.6), cluster 1 contains short build-orders that use mostly just marines. These could be rushes or possibly victims rushes. The build-orders in cluster 2 are mostly mid-length and start with marines but quickly move to either a mix of vultures and siege tanks or just siege tanks. Cluster 3 has build-orders of varying lengths and they tend to contain lots of goliaths (along with drop ships). The cluster 3 build-orders seem to represent strategies that rely on quickly creating a large amount of goliaths, along with dropships to move them around the map. Cluster 4 is the largest cluster and has the same characteristics as cluster 4 for Protoss. Build-orders are long and the cluster is too large to easily see high-level similarities.

3.5 Conclusion

In this chapter we presented a method for clustering strategies for Real-Time Strategy games. Strategies are taken to be build-orders and build-orders are just sequences of characters. We use sequence alignment algorithms to provide a family of similarity metrics between build-orders, and show how a metric can be developed using the game StarCraft. We use a agglomerative hierarchical clustering technique to cluster build-orders and show how payoff matrices can be built using replay data. Clusters show some cognitive coherence, which is encouraging. Our work is preliminary in the sense that it has not been applied to actual RTS game playing and future works include trying different clustering techniques, experimenting with custom cost functions, and using payoff matrices to influence in-game decision making.

Chapter 4

State Evaluation

When an experienced humans play RTS games, they often have a good sense of when they are winning or losing. The goal of this chapter is to identify features of an RTS match that can be used to predict which player will win. We provide an empirical study using StarCraft replays and show the predictive power of our technique. We also present a metric for estimating a player's skill at micro-managing their units (which can also be used as a predictive feature). The work in this chapter is being published at AIIDE 2014 [58].

4.1 Data

For our data set, we used a collection of replays collected by Synnaeve [31]. Since professional tournaments usually only release videos of matches and not the replay files themselves, the replays were taken from amateur ladders (GosuGamer (http://www.gosugamers.net), ICCUP (http://iccup.com/en/starcraft), and TeamLiquid (http://www.teamliquid.net)). Synnaeve et al. released both the replay files themselves and parsed text files (http://emotion.inrialpes.fr/people/synnaeve/), but we decided to write our own parser (github.com/gkericks/SCFeatureExtractor), because of the specific nature of our task and to reduce the possible sources of error. Synnaeve et al. collected ∼8000 replays of all different faction match-ups, but we decided to focus on just the ∼400 Protoss versus Protoss matches because the Protoss faction is the most popular faction among AI systems and our in-house StarCraft system plays Protoss.

We wrote our replay parser in C++ and it uses BWAPI to extract information from replay file which use a compact representation that needs to be interpreted by the StarCraft engine itself. BWAPI then lets one inject a program like our parser into

the StarCraft process. The parser outputs two text files: one with feature values for every player and one with information about the different battles that happened. The first file also includes information about the end of the game, including information about who won the game (which is not always available from the replays) and the final game score, which is computed by the StarCraft game engine. The exact way that the game score is computed is obfuscated by the engine, and the score could not be computed for the opponent in a real game, because of the imperfect information nature of StarCraft, so the game score itself is not a viable feature. In the first file, feature values are extracted for each player according to some period of frames, which is an adjustable parameter in the program.

4.2 Battles

Our technique for identifying battles, as shown by Algorithm 4, is very similar to one presented in [31]. The main difference is what information is logged. Synnaeve et al. were concerned with analyzing army compositions, but we want to be able to actually recreate the battles (the details of which are explained in Section 4.4.4). In Algorithm 4, ON_ATTACK is function that gets called when a unit is executing an attack (during a replay) and UPDATE is a function called every frame. All ON_ATTACK instances for a single frame are called before UPDATE is called. When a new unit is encountered in both the ON_ATTACK and UPDATE functions, the unit's absolute position, health, shield, and the time the unit entered the battle are recorded. When the battle is found to be over (which happens when one side is out of units or no attack actions have happened for some threshold time Δ), the health and shields are recorded for each unit, as well as the time that the battle ended at. Another significant difference is that we start battles based on attacks actions happening (whereas Synnaeve et al. start battles only when a unit is destroyed).

4.3 Preprocessing

From viewing the replays themselves, it became apparent that some of the replays would be problematic for our type of analysis. Some games contained endings where the players would appear to be away from their computers, or where one player was obviously winning but appeared to give up. Such discrepancies could

Algorithm 4 Technique for extracting battles from replays. Note that UPDATE is called every frame and ON_ATTACK is called when a unit is found to be in an attacking state.

```
Global: List CurrentBattles = []
Global Output: List Battles = []
function ON_ATTACK(Unit u)
    if u in radius of a current battle then
        return
    end if
    U ← Units in MIN_RAD of u.position
    B ← Buildings in MIN_RAD of u.position
    if U does not contain Units from both players then
        return
    end if
    Battle b
    b.units ← U ∪ B
    UPDATE_BATTLE(b, U)
    CurrentBattles.add(b)
end function
function UPDATE
    for all b in CurrentBattles do
        U ← getUnitsInRadius(b.center, b.radius)
        if U = ∅ then
            end(b); continue
        end if
        b.units ← U                                   ▷ also log unit info
        if ∃ u ∈ U such that ATTACK(u) then
            UPDATE_BATTLE(b,U)
        end if
        if b.timestamp−CurrentTime() ≥ Δ then
            end(b); continue;
        end if
    end for
    move ended battles from CurrentBattles to Battles
end function
```

cause mis-labeling of our data (in terms of who won each game), so we chose to filter the data based on a set of rules. Table 4.1 shows how many replays were discarded in each step.

When extracting the features from the replays BWAPI has two flags (of interest) that can be true or false for each player: *isWinner* and *playerLeft*. If *isWinner* is present the game is kept and that player is marked as the winner. If *isWinner* is not present, then two things are considered: the *playerLeft* flags (which come with

Algorithm 4 continued

```
function UPDATE_BATTLE(Battle b, UnitSet U)
    center ← average(u.position : u ∈ U)
    maxRad ← 0
    for u ∈ U do
        rad ← distance(u, center) + range(u)
        if rad ≥ maxRad then
            maxRad ← rad
        end if
    end for
    b.center ← center
    b.radius ← maxRad
    b.timestamp ← CurrentTime()
end function
```

Games	Number of Games
Original	447
Kept	391
No Status Close Score	30
Conflict Type A	24
Conflict Type B	1
Corrupt	1

Table 4.1: A breakdown of how many games were discarded

a time-stamp denoting when the player left) and the game score. The game score is a positive number computed by the StarCraft engine. If neither player has a *playerLeft* flag, then we look at the game score. If the game score is close (determined by an arbitrary threshold; for this work, we used a value of 5000), the game is discarded. We chose a relatively large threshold because we want to be confident that the player we picked to be the winner is actually the winner. Otherwise, the player with the larger score is selected as the winner. If there is one *playerLeft* flag, the opposite player is taken as the winner, unless that conflicts with the winner as suggested by the game score, in which case the game is discarded (Conflict Type A). If there are two *playerLeft* flags, the player that left second is taken as the winner unless that conflicts with the winner as suggested by the game score, in which case the game is discarded (Conflict Type B). If a replay file was corrupted (i.e. caused StarCraft to crash at some point) we discarded it as well.

4.4 Features

After the replays are parsed and preprocessed, we represent the data in the form of a machine learning problem. For our matrix X the examples come from the sample states from the games in the Protoss versus Protoss data set. States were taken every 10 seconds for each replay, so each game gave several examples. Since the match-up is symmetric, we let the features be in terms of the difference between feature values for each player and put in two examples for every game state. For example, if player A has D_A Dragoons and player B has D_B Dragoons and player A wins the game, then one example will have $D_A - D_B$ as the number of Dragoons and a target value of 1 and the other example will have $D_B - D_A$ as the number of Dragoons and a target value of 0. This ensures that the number of 1s and 0s in Y is equal and the learned model is not biased toward either target value.

4.4.1 Economic

Economic features are features that relate to a player's resources. In StarCraft there are two types of resources, minerals and gas. For both resource types we include the current amount that the player has R_{cur} (unspent). A forum post on TeamLiquid (http://www.teamliquid.net/forum/starcraft-2/266019-do-you-macro-like-a-pro) showed that the ladder StarCraft 2 players were in was correlated with both average unspent resources U and income I. We included both quantities as features in our model. Average unspent resources is computed by taking the current resources a player has at each frame, and dividing that total by the number of frames. Income is a rate of the in-flow of resources and can be computed as simply the total resources R_{tot} divided by the time passed (T). We stress that it is important that these features are included together because an ideal player will keep their unspent resources low (showing that they spend quickly) and keep their income high (showing that they have a good economy).

$$U = (\sum_{t \leq T} R_{cur})/T$$

$$I = \frac{R_{tot}}{T}$$

4.4.2 Military

Every unit and building type has its own feature in the model. As discussed previously, these features are differences between the counts of that unit type for each player. Features for the Scarab and Interceptor unit types were not included, as those units are used as ammo by Reavers and Carriers respectively and the information gained by their existence would already be captured by the existence of the units that produce them. In an earlier version of the model features for research and upgrades were included as well, but we decided to remove them from the final version due to scarcity and the balanced nature of the match-up. The set of all unit count features is UC.

4.4.3 Map Coverage

We chose a simple way of computing map coverage, as a way of approximating the amount of the map that is visible to each player. Each map is divided into a grid, where tiles contain 4 build tiles (build tiles are the largest in-game tile). A tile is considered occupied by a player if the player has a unit on the tile. Tiles can be occupied by both players. Our map coverage score is then a ratio of the total number of occupied tiles to the total number of tiles. For the feature included in the final version of the model, we just count units (not buildings) and only consider tiles that have walkable space (so the score is map specific). This score can then be computed for each player at a given state, and the included feature, MC, is the difference of those two scores. If P is the set of walkable tiles, then for player p MC can also be formalized as:

$$MC(p) = \sum_{pos \in P} f(pos, p)$$

$$f(pos, p) = \begin{cases} 1 & \text{if } pos \text{ is occupied by } p \\ 0 & \text{otherwise} \end{cases}$$

4.4.4 Micro Skill

Skill is an abstract concept and different aspects of a player's skill can be seen by observing different aspects of a game. The combat portion of the game (also known as micro-managing) takes a very specific type of skill. We devised a feature to capture the skill of a player at the micro portion of the game. The feature makes use of the ideas in [43], where a scripted player is used as a baseline to compare a

player's performance against. We grab battles (as shown in Section 4.2), play them out using a StarCraft battle simulator (*SparCraft*, developed by David Churchill (code.google.com/p/sparcraft), and compare the results of the scripts to the real battle outcomes. We use a version of *SparCraft* edited to support buildings as obstacles, as well as units entering the battle at varying times.

We used a scripted player as the baseline. For this work, we use the *NOK-AV* (No-OverKill-Attack-Value) script and a version of the LTD2 evaluation function to get a value from a battle [7]. *NOK-AV* has units attack an enemy in range with the highest damage-per-frame / hit-points, and has units switch targets if the current target has been assigned a lethal amount of damage already. Note that although we include buildings in the battles, buildings are not acknowledged explicitly by the script policy, and thus are just used as obstacles.

We need a way of comparing the outcome of the real battle to that of the baseline, in a way that can be used as a feature in our model. For a single player, with a set of units U, the Life-Time-Damage-2 (LTD2) [59] score is:

$$\text{LTD2}_{start}(U) = \sum_{u \in U} \sqrt{\text{HP}(u)} \cdot \text{DMG}(u)$$

LTD2 is an evaluation function which favours having multiple units to single units given equal summed health and rewards keeping units alive that can deal greater damage quicker. LTD2 is sufficient for calculating the army value at the end of the battle, but since units can enter the battle at varying times, we need a way of weighting the value of each unit. Let T be the length of the battle and st(u) be the time unit u entered the battle (which can take values from 0 to T). Then:

$$\text{LTD2}_{end}(U) = \sum_{u \in U} \frac{T - \text{st}(u)}{T} \cdot \sqrt{\text{HP}(u)} \cdot \text{DMG}(u)$$

Let one of the players be the player (P) and the other to be the opponent (O). Which player is which is arbitrary, as there are two examples for each state (where both possible assignments of player and opponent are represented). Let P_{out} and P_s be the player's units at the end and the start of the battle respectively, and O_{out} and O_s to be the opponent's units at the end and the start of the battle respectively. The value for the battle is then:

$$V^P = (\text{LTD2}_{end}(P_{out}) - \text{LTD2}_{end}(O_{out})) - (\text{LTD2}_{start}(P_s) - \text{LTD2}_{start}(O_s))$$

To get a baseline value for the battle, we take the initial army configuration for each player (including unit positions and health) and play out the battle in the simulator until time T has passed. At that point, let P_β and O_β be the remaining units for both player and opponent respectively. Then the value given by the baseline player is:

$$V^\beta = (\text{LTD2}_{end}(P_\beta) - \text{LTD2}_{end}(O_\beta)) - (\text{LTD2}_{start}(P_s) - \text{LTD2}_{start}(O_s))$$

For a particular state in a game where there have been n battles with values $V_1^P, \ldots, V_n^P$ and $V_1^\beta, \ldots, V_n^\beta$, we can get a total battle score (β_{avg}) by:

$$\beta_{tot} = \sum_{i=1}^{n} (V_i^P - V_i^\beta)$$

$$\beta_{avg} = \frac{\beta_{tot}}{n}$$

Note that β_{tot} the $(\text{LTD2}_{start}(P_s) - \text{LTD2}_{start}(O_s))$ parts for each V_i will cancel out. They are left in the definitions above because we can also represent the feature as a baseline control variate [43]:

$$\beta_{var} = \frac{1}{n} \sum_{i=1}^{n} (V_i^P - \frac{\widehat{\text{Cov}}[V_i^P, V_i^\beta]}{\widehat{\text{Var}}[V_i^P]} \cdot V_i^\beta)$$

We then use β_{var} as a feature in our model. We also devise β_{avg} and β_{var} as estimates of a player's skill at the micro game. Although we do not explore the idea experimentally here, we maintain that β_{var} could be used in an RTS AI system's decision making process: for high values (showing that opponent is skilled) the system would be conservative about which battles it engaged in, and for low values (showing that the opponent is not skilled) the system would take more risks in hopes to exploit the opponent's poor micro abilities.

4.4.5 Macro Skill

In contrast to the micro game, the macro game refers to high-level decision making, usually affecting economy and unit production. The macro game is much more complicated (and it encompasses the micro game) and we do not have a scripted player for the macro game, so the baseline approach to skill estimation does not apply. Instead, we have identified features that suggest how good a player is at managing the macro game. The first, inspired by [60] is the number of frames that

supply is maxed out for SF. Supply is an in-game cap on how many units a player can build. They must construct additional structures to increase the supply cap (S_{max}). Thus, if a player has a large amount of frames where their supply S_{cur} was maxed out, it shows that the player is bad at planning ahead.

$$SF = \sum_{t \leq T} f(t)$$

$$f(t) = \begin{cases} 1 & \text{if } S_{cur} = S_{max} \text{ at time t} \\ 0 & \text{otherwise} \end{cases}$$

The other feature is the number of idle production facilities PF. A production facility is any structure that produces units and if a player has a large amount of production facilities which are not actively producing anything, it suggests that the player is poor at managing their unit production. A third feature is the number of units a player has queued Q. Production facilities have queues that units can be loaded into, and it is considered skillful to keep the queues small (since a player must pay for the unit the moment it is queued). However, both PF and Q could never be used as part of a state evaluation tool, since that information is not known even with a perfect information assumption.

4.5 Learning

This chapter uses logistic regression as the central learning technique. The data is represented as a matrix X with n examples (rows) and k features (columns). Each row has a corresponding target value, 1 for a player win 0 for a player loss, shown in the column vector Y. The logistic regression algorithm takes X and Y and gives a column vector K of k weights such that

$$X \cdot K = Y'$$

$$T(g(Y')) \approx Y$$

$$g(s) = \frac{1}{1 + e^{-s}}$$

Y' *predicts* Y and the weights K can be used to predict new targets given new examples by applying the weights to the features as a linear combination and then

putting those sums through a sigmoid function (g). The result is a response value for each example, i.e. a real number between 0 and 1 that can be thresholded (e.g., T = Heaviside step function) to act as a prediction for the new examples.

4.6 Feature Set Evaluation

For the purposes of experimentation, we did 10-fold cross validation on games. Recall that our example matrix X is made up of examples from game states from different games at different times. If we permuted all the examples and did cross validation we would be training and testing on states that are from the same games (which introduces a bias). So instead we chose to split the games into 10 different folds. When we leave a particular fold out for testing, we are leaving out all the states that were taken from the games in that fold. So for each fold, logistic regression is ran on all the other folds giving a set of feature weights. Those feature weights can then be used as the weights for a linear combination of the features of each of the examples in the testing fold and then putting those sums through a squashing function (sigmoid function). The result is a response value for each example i.e. a real number between 0 and 1 that acts as a prediction for the test examples.

We report two different metrics for showing the quality of our model. The first is accuracy, which is just the proportion of correct predictions (on the test folds) to the total number of examples. Predictions are taken from the response values by thresholding them. We used a threshold value of 0.5, which is standard. The second is the average log-likelihood [61]. For a single test example, whose actual target value is given by y and whose response value is r, the log-likelihood is defined as follows:

$$L(y, r) = y \cdot \log(r) + (1 - y) \cdot \log(1 - r)$$

We report the average log-likelihood across examples. Values closer to zero indicate a better fit.

Through experimentation we want to answer the following questions: 1) Can our model be used to accurately predict game outcomes? 2) Does adding the skill features to our model improve the accuracy? 3) What times during a game is our model most applicable to?

To explore these questions we tested a few different feature subsets on examples

from different times in the games. Especially in the early game, result prediction is a very noisy problem because there is little evidence in the early game as to who is going to win because the important events of the game have not happened yet. We decided to experiment by running separate cross validation tests just using examples that fall within certain time intervals. For time intervals, we use 5 minute stretches and just include any example with a time-stamp greater than 15 minutes together. Note that not all games are the same length, so for the later time intervals not all games are included. 5 minute interval lengths were chosen because we wanted to have a reasonable amount of examples in each interval while still doing a fine-grained analysis. Table 4.2 shows how examples were divided based on time-stamps.

Time (min)	Games	Examples
0-5	391	23418
5-10	386	22616
10-15	364	19836
15-20	289	14996
20-	211	31060

Table 4.2: A breakdown of how examples were split by time-stamp

Table 4.3 shows the performance of using the feature sets individually. The map coverage feature MC preforms very well in the late game because a good MC value near the end of the game is the result of having a good economy earlier in the game and having a strong army. In general, prediction rates improve in the later game stages. β_{var} has a drop in accuracy in the late game because many of the battles at that stage of the game include unit types which our simulator does not support and so late game battles (which are important to the game outcome) are ignored. Table 4.3 also shows feature sets being added, culminating in the full model in line C. Notice that the skill features make a difference in the late game, due to there being differences in skill being more noticeable as the game progresses (SF takes to grow, as players need to make mistakes for it to be useful). When choosing intervals we had problems with over-fitting. UC especially is prone to over-fitting if the training set is too small. Table 4.4 shows how we tested with larger training sets to avoid over-fitting. Results are overall slightly lower because the early intervals are trained with all or most of the timestamps, and examples from 20- are never tested on.

Features	0-5	5-10	10-15	15-
R_{cur},I,U	54.42 (-0.686)	57.76 (-0.672)	62.98 (-0.647)	64.17 (-0.625)
UC	51.96 (-0.712)	57.84 (-0.682)	66.67 (-0.705)	66.46 (-0.644)
MC	51.27 (-0.693)	55.20 (-0.685)	61.45 (-0.657)	71.39 (-0.561)
β_{var}	50.23 (-0.693)	53.25 (-0.690)	55.09 (-0.690)	52.82 (-0.690)
SF, PF, Q	51.26 (-0.695)	49.96 (-0.695)	51.75 (-0.694)	54.97 (-0.709)
A	53.91 (-0.708)	58.81 (-0.680)	66.36 (-0.712)	69.22 (-0.613)
B	54.05 (-0.708)	58.66 (-0.681)	66.44 (-0.712)	69.87 (-0.608)
C	53.81 (-0.710)	58.72 (-0.681)	66.41 (-0.708)	72.59 (-0.587)

Table 4.3: Individual feature (group) and feature set prediction performance reported as accuracy(%) (avg L) in each game time period; A = economic/military features R_{cur}, I, U, UC; B = A + map control feature MC; C = B + skill features β_{var}, SF, PF, Q

Feature Set	0-5	5-10	10-15	15-20
R_{cur},I,U	53.75 (-0.6875)	58.85 (-0.6708)	62.82 (-0.6510)	60.23 (-0.6562)
UC	52.03 (-0.6936)	58.43 (-0.6735)	65.76 (-0.6329)	63.96 (-0.6516)
MC	51.27 (-0.6943)	55.20 (-0.6872)	61.45 (-0.6588)	64.02 (-0.6385)
β_{var}	50.23 (-0.6931)	53.25 (-0.6896)	55.24 (-0.6899)	56.14 (-0.6868)
SF, PF, Q	52.02 (-0.6925)	50.74 (-0.6939)	52.82 (-0.6916)	55.21 (-0.6857)
A	53.19 (-0.6917)	58.74 (-0.6726)	65.28 (-0.6367)	63.58 (-0.6612)
B	52.60 (-0.6916)	58.56 (-0.6727)	64.89 (-0.6377)	63.99 (-0.6617)
C	52.73 (-0.6914)	58.70 (-0.6669)	65.77 (-0.6267)	65.23 (-0.6510)

Table 4.4: Feature set prediction performance [accuracy(%) (avg L)]; If time interval is [k,l] training is done on examples in [k,∞) and tested on examples in [k,l]

As an additional evaluation of the performance of the feature set for predicting game outcome, we provide an experiment where only terminal states are evaluated. The goal is to show that our model gets high accuracy on terminal states. For this experiment, we took terminal states to be the last examples recorded for each game. Tables 4.5 and 4.6 show that as long as the training set includes mid or late game examples, the predictor has very high accuracy.

0-5	5-10	10-15	15-
55.75	82.90	85.71	97.23

Table 4.5: Accuracy(%) on terminal states with training done on the provided time interval

0-	5-	10-	15-
98.21	98.19	97.53	97.23

Table 4.6: Accuracy(%) on terminal states with training done on the provided time interval

4.7 Battle Metric on Tournament Data

We use the battle metric β as a feature in our feature set for state evaluation. β is an estimator for a player's skill at the micro game. By micro game, we are referring to the management of units on an individual or squad-based level in isolated skirmishes. We believe that a player's skill at the micro game is correlated with their overall skill. This is a difficult thing to show by merely using the metric as a feature in the result prediction problem. In an individual match, there are many other factors that can affect the eventual result than simply a player's micro skill. Over a large amount of battles irregularities would be less noticeable and a general notion of the player's skill would become apparent. To show the merit of β as a skill estimator a large amount of battles for each player would be have to be available. If that was available then β could be compared to some other measure of the player's success to show that β and a player's skill are related. The Synnaeve data-set could have potentially been used for this if the same players were found in multiple matches in the dataset. We found that most players could only be located in one or two matches, so this was not the case. The Synnaeve dataset could still have been useful if some other measure of the player's skill was available. We looked on the online ladders that the replay files were downloaded from to see if player standings from different points in time were available, although this was also not the cases.

To show the effectiveness of β as a skill estimator we decided to use replay data from the 2013 AIIDE StarCraft AI Tournament [62]. AIIDE (Artificial Intelligence in Digital Entertainment) is a conference that holds a competition for StarCraft AI annually. The competition is held at the University of Alberta and is the top competition for open-source StarCraft AI systems. Replays from the tournament are freely available online, along with the tournament results. In the 2013 competition eight different systems were entered. Each system played against every other system twenty times on each map. Ten different maps were used. Win percentages

were tallied and systems were ranked based on their win rate.

The tournament data suites our method for calculating β with SparCraft for multiple reasons. Players (in this case AI systems) are easily track-able across a large amount of matches. Rankings for the players are readily available. As discussed earlier, SparCraft currently has limited support for the different unit types/features found in StarCraft. Spell-casters, flying units, and any other units with special features (beside medics) cannot be simulated in SparCraft. In the analysis of the Synnaeve data, battles that included such units were simply dropped. We do the same thing here, but in general that is less of an issue for AI systems. The matches in the tournament tend to end faster than human matches (which decreases the amount of late-game units encountered) and for tactical simplicity many systems favour simple units such as Zealots and Dragoons (which are supported by SparCraft). The downside to the tournament data is that any analysis is done using programs instead of humans. Humans are capable of playing RTS games adaptively and effectively and still far better than AI systems at RTS. Thus findings done with the tournament dataset cannot be claimed to be discovered on professional quality RTS replays.

For experimentation, we parsed the tournament replays, extracting the battles from each match. Battles were defined by algorithm 4 (we used our parser). Battles were replicated in SparCraft using the NOK-AV script and the results were recored. We then could compute β individually for each player. The sum term in β sums over the set of all the battles kept for the player being processed. Players can then be ranked in terms of β and compared to the original tournament standings. The tournament ranking is given in Table 4.7. The order the systems are listed in refers to the ranking, and win percentages are also presented for each system.

UAlbertaBot	82.43%
Skynet	72.77%
Aiur	60.29%
Ximp	55.29%
Xelnaga	49.96%
ICEStarCraft	47.82%
Nova	27.47%
BTHAI	3.93%

Table 4.7: Ranking from AIIDE 2013 StarCraft Competition (program name and win percentage)

We calculated values for both β_{avg} and β_{var}. The ranking found using β_{avg} can be found in Table 4.8 and the ranking found using β_{var} can be found in Table 4.9 (the order shows the ranking, β values are listed). The rankings found using β_{avg} and β_{var} are very similar (with the only difference being Xelnaga being ranked slightly higher in the β_{var} ranking). We speculate this is because the number of data samples for each player is high enough that the variance reduction method brought in by β_{var} is not necessary (i.e. with enough data points the increase in variance caused by outliers will be lower).

Nova	7.65
UAlbertaBot	3.30
Aiur	1.01
ICEStarCraft	-0.026
Ximp	-1.91
BTHAI	-3.03
Skynet	-4.61
Xelnaga	-5.60

Table 4.8: Ranking using β_{avg}

Nova	7.59
UAlbertaBot	1.97
Aiur	0.85
ICEStarCraft	0.01
Ximp	-1.79
Xelnaga	-2.99
BTHAI	-3.13
Skynet	-4.51

Table 4.9: Ranking using β_{var}

The rankings provided by using the β terms show some similarities to the original tournament ranking and some differences. UAlbertaBot and Aiur are high up in both the original and β rankings. BTHAI and Xelnaga are both low in both the original and β rankings. ICEStarCraft is higher in the β rankings (when compared to the original rankings) and Ximp is slightly lower (as a result of ICEStarCraft being higher). The biggest changes between the original rankings and the β rankings are that Skynet and Nova basically switched spots with each other (when moving from the original rankings to the β rankings). Skynet poor ranking could merely

have been because of noise (although we could not identify a more concrete reason). Although Nova preformed poorly in the tournament overall, it is known for having effective micro tactics. So it is possible that Nova could have preformed well at battles (which was expected) and lost most of its matches because of mistakes made in other areas (such as the macro game). Overall, we are pleased that the general ranking formed using β is similar to the original tournament ranking, as this suggests that β works as a skill estimator.

4.8 Conclusion

In this chapter we attempted to solve the problem of predicting which player will win in an RTS match. We presented a set of features that can be used to predict the outcome of a match, and showed that our feature set has $> 70\%$ accuracy in the later stages of a match. Of special interest is the battle skill estimator β, which can be used as a feature in predicting a match's winner. Search is a powerful game-playing technique, and as search algorithms become scaled to the global RTS game, state evaluation will prove useful in pruning search trees and informing decision making.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this thesis, we provided techniques for solving two machine-learning problems in RTS games. First, we presented a scheme for identifying groups of strategies present in replay data. We showed how similarity metrics between strategies can be engineered using sequence alignment algorithms and a small amount of expert knowledge. We also gave examples of how the methods can be applied to real data, using the RTS game StarCraft, and built a few different payoff matrices using the results of the replays. We found that non-symmetric match-ups make for more interesting payoff matrices, and that pruning away clusters with a small number of examples can lead to more interesting payoff matrices.

We also gave a feature-set based solution to the problem of predicting a match outcome. Our results show that prediction of the game winner in an RTS game domain is possible, although the problem is noisy. Prediction is most promising in later game stages. This gives hope for quick evaluation in future search algorithms. Of particular interest is the use of simulation for providing a baseline to estimate a player's skill at the micro game against. We used the metric as a feature in prediction, with moderate success (the simulator currently has limited functionality). We also applied the metric to skill estimation in a AI tournament setting and used the values to rank the players based on their micro skill. We saw similarity between the skill estimation rankings and the original tournament standings, with a few interesting discrepancies.

5.2 Future Work

The main direction future works should be in is application. The work presented in this thesis has been applied to real data, so the findings are not purely theoretical, but the application of the results themselves to develop more intelligent RTS AI has yet to be done. For the strategy clustering problem, the first step of applying the results to AI system development will be to map strategy clusters to implementable strategy policies. This could be done by simply looking at cluster elements and having an expert describe the policies the strategies represent (the policies could then be hard-coded for the program). More sophisticated methods could be pursued as well, such as identifying representative elements in each cluster automatically and using those build orders in some way in the AI system (this would limit the adaptability of the system, but could possibly be effective if paired with a build-order search algorithm). After this is done the strategies could be loaded into an AI system along with a strategy detection program (such as Synnaeve's or Dereszynski's). Then when an enemies strategy is predicted, the system could employ the strategy with the highest corresponding payoff in the payoff matrix.

Future work for the results prediction problem also involves application. Eventually we would like to see the evaluator used as part of a search algorithm (for pruning probably). The micro skill estimator could be used in an AI system to affect its decision making regarding attacking or retreating. The skill estimator itself could be extend to incorporate multiple estimates. We only used one script to get the baseline values, but multiple baseline scripts could be used to get multiple estimates. The baseline player doesn't have to be a rigid script either, searches could be used (even stochastic algorithms are acceptable as long as multiple runs are averaged over).

Further work could be done on the results predictor. The feature set we provided here could be expanded to incorporate more complex features. Possible areas of to explore could be incorporating high-level descriptions of the strategy the players are using, as features, or modeling the tech tree explicitly (although we feel that using unit counts supersedes the information gain from using tech trees). Further work will have to be done on adapting our model to the imperfect information case for the model to be effective in actual application. We suggest that this could be as simple as estimating the unknown quantities using visible evidence.

Bibliography

[1] M. Buro, "Real-time strategy games: A new AI research challenge," in *IJCAI 2003*, pp. 1534–1535, International Joint Conferences on Artificial Intelligence, 2003.

[2] M. Buro and D. Churchill, "Real-time strategy game competitions," *AI Magazine*, vol. 33, no. 3, pp. 106–108, 2012.

[3] S. Ontañón, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill, and M. Preuss, "A survey of real-time strategy game AI research and competition in StarCraft," *TCIAIG*, 2013.

[4] E. Dereszynski, J. Hostetler, A. Fern, T. D. T.-T. Hoang, and M. Udarbe, "Learning probabilistic behavior models in real-time strategy games," in *Artificial Intelligence and Interactive Digital Entertainment (AIIDE)* (AAAI, ed.), 2011.

[5] D. Churchill and M. Buro, "Portfolio greedy search and simulation for large-scale combat in StarCraft," in *IEEE Conference on Computational Intelligence in Games (CIG)*, pp. 1–8, IEEE, 2013.

[6] S. Ontañón, "Experiments with game tree search in real-time strategy games," *arXiv preprint arXiv:1208.1940*, 2012.

[7] D. Churchill, A. Saffidine, and M. Buro, "Fast heuristic search for RTS game combat scenarios," in *AI and Interactive Digital Entertainment Conference, AIIDE (AAAI)*, pp. 112–117, 2012.

[8] M. Buro and T. Furtak, "On the development of a free RTS game engine," in *GameOn Conference*, pp. 23–27, Citeseer, 2005.

[9] J. McCoy and M. Mateas, "An integrated agent for playing real-time strategy games.," in *AAAI*, vol. 8, pp. 1313–1318, 2008.

[10] D. E. Knuth and R. W. Moore, "An analysis of alpha-beta pruning," *Artificial Intelligence*, vol. 6, no. 4, pp. 293–326, 1976.

[11] M. Buro, "ProbCut: An effective selective extension of the Alpha-Beta algorithm," *ICCA Journal*, vol. 18, no. 2, pp. 71–76, 1995.

[12] M. Enzenberger, M. Muller, B. Arneson, and R. Segal, "Fuego—an open-source framework for board games and go engine based on Monte Carlo tree search," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 2, no. 4, pp. 259–270, 2010.

[13] G. Van den Broeck, K. Driessens, and J. Ramon, "Monte Carlo tree search in poker using expected reward distributions," in *Advances in Machine Learning*, pp. 367–381, Springer, 2009.

[14] M. Chung, M. Buro, and J. Schaeffer, "Monte Carlo planning in RTS games," in *IEEE Symposium on Computational Intelligence and Games (CIG)*, 2005.

[15] R.-K. Balla and A. Fern, "UCT for tactical assault planning in real-time strategy games," in *IJCAI*, pp. 40–45, 2009.

[16] L. Kocsis and C. Szepesvari, "Bandit based Monte-Carlo planning," in *Proceedings of the European Conference on Machine Learning*, pp. 282–293, 2006.

[17] T. Furtak and M. Buro, "On the complexity of two-player attrition games played on graphs," in *Proceedings of the Sixth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2010* (G. M. Youngblood and V. Bulitko, eds.), (Stanford, California, USA), Oct. 2010.

[18] F. Schadd, S. Bakkes, and P. Spronck, "Opponent modeling in real-time strategy games," in *GAMEON*, pp. 61–70, 2007.

[19] J.-L. Hsieh and C.-T. Sun, "Building a player strategy model by analyzing replays of real-time strategy games," in *IJCNN*, pp. 3106–3111, 2008.

[20] D. W. Aha, M. Molineaux, and M. J. V. Ponsen, "Learning to win: Case-based plan selection in a real-time strategy game," in *ICCBR*, pp. 5–20, 2005.

[21] K. Mishra, S. Ontañón, and A. Ram, "Situation assessment for plan retrieval in real-time strategy games," in *ECCBR*, pp. 355–369, 2008.

[22] S. Ontañón, K. Mishra, N. Sugandh, and A. Ram, "Learning from demonstration and case-based planning for real-time strategy games," in *Soft Computing Applications in Industry* (B. Prasad, ed.), vol. 226 of *Studies in Fuzziness and Soft Computing*, pp. 293–310, Springer Berlin / Heidelberg, 2008.

[23] P. Cadena and L. Garrido, "Fuzzy case-based reasoning for managing strategic and tactical reasoning in StarCraft," in *MICAI (1)*, pp. 113–124, 2011.

[24] M. Sharma, M. Holmes, J. Santamaria, A. Irani, C. L. Isbell, and A. Ram, "Transfer learning in real-time strategy games using hybrid CBR/RL," in *International Joint Conference of Artificial Intelligence, IJCAI*, 2007.

[25] B. G. Weber and M. Mateas, "A data mining approach to strategy prediction," in *IEEE Symposium on Computational Intelligence and Games (CIG)*, 2009.

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

[27] G. Synnaeve and P. Bessière, "A Bayesian model for plan recognition in RTS games applied to StarCraft," in *Proceedings of the Seventh Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2011)* (AAAI, ed.), Proceedings of AIIDE, (Palo Alto, États-Unis), pp. 79–84, Oct. 2011.

[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*, pp. 281–288, 2011.

[29] G. Synnaeve and P. Bessiere, "Special tactics: a Bayesian approach to tactical decision-making," in *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, pp. 409–416, 2012.

[30] G. Synnaeve and P. Bessiere, "A Bayesian model for RTS units control applied to StarCraft," in *Proceedings of IEEE CIG 2011*, (Seoul, Corée, République De), p. 000, Sept. 2011.

[31] G. Synnaeve and P. Bessiere, "A dataset for StarCraft AI & an example of armies clustering," in *AIIDE Workshop on AI in Adversarial Real-time games 2012*, 2012.

[32] B. G. Weber, M. Mateas, and A. Jhala, "A particle model for state estimation in real-time strategy games," in *Proceedings of AIIDE*, (Stanford, Palo Alto, California), p. 103–108, AAAI Press, AAAI Press, 2011.

[33] S. Wender and I. Watson, "Applying reinforcement learning to small scale combat in the real-time strategy game StarCraft:Broodwar," in *CIG (IEEE)*, 2012.

[34] Q. Gemine, F. Safadi, R. Fonteneau, and D. Ernst, "Imitative learning for real-time strategy games," in *CIG (IEEE)*, 2012.

[35] N. Othman, J. Decraene, W. Cai, N. Hu, and A. Gouaillard, "Simulation-based optimization of StarCraft tactical AI through evolutionary computation," in *CIG (IEEE)*, 2012.

[36] J. M. Traish and J. R. Tulip, "Towards adaptive online RTS AI with NEAT," in *CIG (IEEE)*, 2012.

[37] P. Yang, B. Harrison, and D. L. Roberts, "Identifying patterns in combat that are predictive of success in MOBA games," *Proceedings of Foundations of Digital Games 2014*, p. to appear, 2014.

[38] M. Stanescu, S. P. Hernandez, G. Erickson, R. Greiner, and M. Buro, "Predicting army combat outcomes in StarCraft," in *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2013.

[39] T. Avontuur, P. Spronck, and M. van Zaanen, "Player skill modeling in StarCraft II," in *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2013.

[40] J. Long, "Game theoretic and machine learning techniques for balancing games," Master's thesis, University of Saskatchewan, 2006.

[41] D. Churchill, "SparCraft: open source StarCraft combat simulation." `http://code.google.com/p/sparcraft/`, 2013.

[42] D. Churchill and M. Buro, "Incorporating search algorithms into RTS game agents," in *AI and Interactive Digital Entertainment Conference, AIIDE (AAAI)*, 2012.

[43] J. Davidson, C. Archibald, and M. Bowling, "Baseline: practical control variates for agent evaluation in zero-sum domains," in *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*, pp. 1005–1012, International Foundation for Autonomous Agents and Multiagent Systems, 2013.

[44] D. Churchill and M. Buro, "Build order optimization in StarCraft," in *AI and Interactive Digital Entertainment Conference, AIIDE (AAAI)*, pp. 14–19, 2011.

[45] Y.-K. Yu, "Sequence alignment in bioinformatics," in *New Directions in Statistical Physics*, pp. 193–212, Springer, 2004.

[46] E. Sumita, "Example-based machine translation using dp-matching between word sequences," in *Proceedings of the workshop on Data-driven methods in machine translation-Volume 14*, pp. 1–8, Association for Computational Linguistics, 2001.

[47] R. Agrawal and R. Srikant, "Mining sequential patterns," in *Data Engineering, 1995. Proceedings of the Eleventh International Conference on*, pp. 3–14, IEEE, 1995.

[48] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of molecular biology*, vol. 48, no. 3, pp. 443–453, 1970.

[49] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals," in *Soviet physics doklady*, vol. 10, p. 707, 1966.

[50] K. Duraiswamy and V. V. Mayil, "Similarity matrix based session clustering by sequence alignment using dynamic programming," *Computer and Information Science*, vol. 1, no. 3, p. P66, 2008.

[51] P.-N. Tan, M. Steinbach, and V. Kumar, "Cluster analysis: basic concepts and algorithms," *Introduction to data mining*, pp. 487–568, 2006.

[52] I. Gath and A. B. Geva, "Unsupervised optimal fuzzy clustering," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 11, no. 7, pp. 773–780, 1989.

[53] J. C. Dunn, "A fuzzy relative of the isodata process and its use in detecting compact well-separated clusters," 1973.

[54] S. C. Johnson, "Hierarchical clustering schemes," *Psychometrika*, vol. 32, no. 3, pp. 241–254, 1967.

[55] J. H. Ward Jr, "Hierarchical grouping to optimize an objective function," *Journal of the American statistical association*, vol. 58, no. 301, pp. 236–244, 1963.

[56] G. Lance and W. Williams, "A generalized sorting strategy for computer classifications," 1966.

[57] J. S. Farris, "On the cophenetic correlation coefficient," *Systematic Biology*, vol. 18, no. 3, pp. 279–285, 1969.

[58] G. Erickson and M. Buro, "Global state evaluation in StarCraft," in *Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2014 in press.

[59] A. Kovarsky and M. Buro, "Heuristic search applied to abstract combat games," *Advances in Artificial Intelligence*, pp. 66–78, 2005.

[60] J. Belcher, D. Ekins, and A. Wang, "Starcraft 2 oracle," *University of Utah CS6350 Project*, 2013. `http://www.eng.utah.edu/~cs5350/ucml2013/proceedings.html`.

[61] M. E. Glickman, "Parameter estimation in large dynamic paired comparison experiments," *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, vol. 48, no. 3, pp. 377–394, 1999.

[62] D. Churchill, "AIIDE 2013 AIIDE StarCraft AI Competition Report." `http://webdocs.cs.ualberta.ca/~cdavid/starcraftaicomp/report2013.shtml`, 2013.