USING POISSON MARKOV MODELS TO PREDICT GAME STATES IN STARCRAFT

ANDERS HESSELAGER-OLESEN

d1012e13
Department of Computer Science
AAU

Spring 2014

ABSTRACT

This project aims to investigate the feasibility of using mixed observation types in Hidden Markov Models, in an attampt to increase the accuracy of recognising strategies, and predicting future actions in the domain of th real-time strategy game StarCraft. The types of observations in the model will be multinomial and Poisson distributions, and theory for both types of variables will be presented, as will theory for the combination of the two. The data for training the model in the StarCraft domain will be analysed to determine the validity of applyig Poisson distributions to the production of combat units. Finally, experiments will be made to determine the accuracy of predictions made by the model, as well as an evaluation of the most likely path through the state space of the model.

Anders Hesselager-Olesen

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

BIC Bayesian information criterion

BWAPI Brood War API

DAG Directed Acyclic Graph

EM Expectation Maximization

FoW Fog of War

HMM Hidden Markov Model

PHMM Poisson Hidden Markov Model

RTS Real-Time Strategy

SC StarCraft

LIST OF SYMBOLS

X_t	Unobserved variable for the state of a Hidden Markov Model for time step t
X_0	The initial stationary distribution for the model
y_t	Observation for time step t
Y	The full set of observation vectors in a replay
t	The current time step of the sequence
T	The total number of time steps in the sequence
r	The current replay in the collection of replays
R	The total number of replays in the data set
i	The current hidden state of the model
j	The state that the hidden state transitions to
k	The current iteration number of learning
α	The forward message for a given replay
β	The backard message for a given replay
$\gamma_{i,j}$	The transition probability from state *i* to *j*
π	The emission probabilities given a hidden state

1 INTRODUCTION

This project will further investigate the general model developed in the project "Predicting Unit Production in StarCraft using Hidden Markov Models"[7], where a model was created to recognize and predict plans in the domain of the strategy game StarCraft (SC). Plan recognition in SC is interesting since the correct assessment of the opponent's strategy is paramount to the success of a player. The recognition of this strategy is not trivial, since a very precise understanding of the opponent's timing and actions is needed. This report will investigate of the assumptions made in the report "Predicting Unit Production in StarCraft using Hidden Markov Models", regarding the structure of the model, specifically whether multinomial distributions are an accurate representation of all the observations made during a game of SC, or if precision of predictions can be improved by using an alternative distribution for some of the observation variables. In the report "Predicting Unit Production in StarCraft using Hidden Markov Models" models were made with observations grouped in the intervals $0, > 1$, $0, 1, > 2$ and $0, 1 - 2, > 2$. This was an extension of the binary observations originally used for strategy labelling by Dereszynski et al.[11], made in an attempt to refine the predictions and labelling of strategies, since it was believed that strategies might be defined not only by the presence of certain units, but also by their number. Since the previous report, as well as Dereszynski et al. used the game configuration of a protoss player versus a terran player, this configuration will be used in this report as well for the sake of consistency.

1.1 PLAN RECOGNITION

Strategy prediction shares many elements with plan recognition, since, in order to predict a strategy, one must first find a way to determine what describes the strategy – to recognize it. Thus, when the model has learned the patterns of different strategies, inference can be made about future actions[19]. Plan recognition suffers from a degree of uncertainty: Some actions may not be observable, and other actions may happen earlier, or later than in the average case. This leads to a degree of uncertainty that probabilistic models can accommodate, unlike deterministic models, such as decision trees.[19]

Plan recognition exists in two types: *Direct* and *keyhole*. *Direct* plan recognition is the situation where the acting agent intentionally tries to make information available to the observer, as opposed by *keyhole* plan recognition, where the agent is trying its best to occlude its intentions to the observer. Similarly, a distinction is made regarding if the observer has total or incomplete information about the actions taken. In the context of the SC domain the plan recognition is considered to be keyhole, since observed agent has no interest in making information about its strategy, and may even take de-

ceptive actions to suggest that a different strategy is being employed. For the purposes of this project the information level will be total, since model training is based on replays of SC matches, which allow for a total observation of all actions.

1.1.1 *StarCraft*

SC is a Real-Time Strategy (RTS) game that pits a player against one or more opponents. The objective of the game is to amass an armed force sufficient to destroy the opponents' base. SC has three different factions a player can play as: *Protoss*, *Terran* and *Zerg*, each with their own distinct construction options and special rules.

SC has had an extremely active multi-player scene for approximately ten years, with especially 1 versus 1 duels rising to prevalence. In these duels the execution of a player's strategy, and recognition of his opponent's strategy becomes extremely important since SC follows a rock-paper-scissors style of battle. Each unit will have an optimal counter-unit that will defeat it in a pitched battle. For this reason, as well as each unit having an immutable time it takes to create, it becomes extremely important to not only know what armed forces an opponent currently posses, but also being able to predict which units he plans to create in the near future in order to prepare a counter. Different templates for construction of a force, known as strategies, have been formalized in the community, and follow a rigid ordering of construction that has been decided as optimal for what the player is trying to achieve. Learning to recognize an opponent's strategy, and knowing which counter-strategy to employ is fundamental to becoming a good SC player. Predicting strategies is also paramount if one has hopes of becoming a commenter for one of the professional televised leagues of StarCraft's successor, StarCraft II.

1.1.1.1 *Game rules*

The objective of an SC game is to destroy all of an opponent's structures. For this, armed units are used. These armed units are produced from one's own structures. Both production of structures and units costs an investment of resources and then takes a fixed time, depending on the type that is being produced, to finish. Resources are gathered by worker units, which are also responsible for the construction of buildings. A significant part of a strategy is the navigation of the *tech-tree*. The tech tree is approximately a directed acyclic graph, with buildings as nodes. Thus, in order to construct a building, a player must possess the buildings that are present earlier in the graph. Unlike a regular directed acyclic graph, the tech-tree only allows progression through a node, if all the nodes leading to it, have been visited, ie. the buildings have been constructed. Certain buildings are both needed for traversing the tech-tree as well as enabling the production of specific units, where other buildings serve only the purpose of tech-tree traversal.

1.1.2 *Problem statement*

Dereszynski et al. made an expansion of their model themselves [12] where the model is expanded into a dynamic baeysian network. The expanded model did increase the accuracy of predictions, but did also increase the complexity, both computationally and structurally[12]. As part of this project I will investigate if a useful middle ground can be made with a mixed observation hidden markov model, that allows for greater granularity, and thus accuracy, than the model created in "Predicting Unit Production in StarCraft using Hidden Markov Models" without the complexity of the second model created by Dereszynski et al.

The model created in the previous project made the assumption that all observations were multinomial and exhibited the same intervals regardless of the type of observation. This meant that the model treated all observations the same way, regardless of the observation type. This project will investigate the effects of modelling some of the observation variables as Poisson instead of multinomial distributions and investigate if this gives an increase in accuracy when attempting to make predictions. The variability of observations is interesting because certain events may happen only once and others may happen repeatedly during an SC game, or even several times during a single time step of observation. If all observation variables are grouped the same way, some may be grouped too strictly while others may be grouped too loosely. If instead, the observations that have a broad variance were modelled as Poisson distributions, these could be modelled more accurately without adding static to observations that fall within a tighter grouping. This will mean that a HMM that contains both multinomial and distributional observation variables will need both the update rules for multinomial distributions, as well as the lambda parameter for the Poisson distribution.

In this project I will investigate an alternative to the model designed in the project, "Predicting Unit Production in StarCraft using Hidden Markov Models". This alternative model should be able to incorporate both multinomial and Poisson distribution modelled observations, in an effort to increase the accuracy of the model, based on an expectation that building construction and unit construction does not follow the same patterns. Experiments will be performed to investigate the learned models' accuracy in determining the number of living units in a time interval, since the number of armed units the opponent possesses is highly descriptive for the opponent's current intended strategy.

The Poisson distribution will be used as an alternative to the multinomial, since Poisson distributions are often used to descibe the patterns of birth and death in real world populations. It is the hope and expectation that using Poisson distributions for certain of the observations will help to increase the accuracy, since these are expected to follow a similar pattern. The reasoning for having some emissions remain multinomial, is that the construction of buildings is not expected to happen consistently across the length of a game, or the duration of a strategy. Instead, these are expected to happen at the start of a strategy, since buildings help traverse the tech-tree, enabling the production of certain units, which are then expected to be pro-

duced at a relatively constant rate for the duration of the strategy. Theory for the combination of the two types of emissions will have to be devised and implemented.

A Poisson Hidden Markov Model (PHMM) shares most of the features of a regular HMM, described in Chapter 3. In a PHMM the observation variables are not multinomial, but *Poisson distributions*. Instead of an emission probability matrix, an observation variable in a PHMM has a Poisson parameter for each state, denoted by λ, describing the average expected value of the observation variable.

To make a model that accurately shows the current number of units in a time step, a way to extract this information from SC replays is needed to give data for learning. Once a way of extracting this has been found, automation of this process is needed as well. Finally a parser for the extracted data will need to be constructed to convert the data into something that can be used in an HMM. A model will then be constructed that allows for both multinomial and Poisson variables. To determine the accuracy of the model, it will be necessary to not only predict the number of produced units in a time step, but also the number of units killed, since the error margin of production may lead to an erroneous result when predicting the number of live units, even if the number of destroyed units is correct and vice versa. The tests should be performed across a substantial number of replays to ensure the test results' viability in the general case.

2 RELATED WORK

Plan recognition is a well established research field, and as such has given rise to various different approaches. Many of these approaches have been applied to real time strategy games. Some of these approaches descibe the state of the game, where others have a higher focus on the actions taken by the playing agent. In this section some of these approaches will be described and discussed. Since this project builds directly on top of the project "Predicting Unit Production in StarCraft using Hidden Markov Models" much of the related work is the same, and as such has been brought more or less directly into this project.

2.1 CASE BASED PLANNNING

Case based plan recognition tries to infer future actions based on cases the model has previously been exposed to, hence the name. All the previously learned cases are stored in a *case library*, and when predicting future actions, the model refers to the case library and finds the nearest matching case, and uses this as a basis for inference about future actions.[8][13][16][22] The current state of the game has been used as a basis for case based plan recognition in severeal articles. To use the state as a case a definition of states and actions that can bring about state transitions must be made. States are defined by a set of features and thus changing the features that are integrated into the states changes the set of game states. Similarly, the action definitions are dependent on the state definitions and the combinations of states and actions. Defining states and actions for the case library can be problematic: If states and actions are defined manually, the strategies that can be recognized are also, indirectly, defined manually. Dynamic learning of the possible actions for each state type makes for a very complex case library, since the enviroment of an RTS game is also very complex.

2.2 DATA MINING

The data mining approach has been applied to SC by Ben G. Weber and Michael Mateas[23] where decision trees are used to predict the actions of the opponent based on pre-labeled strategies. Each replay category was defined manually using SC domain knowledge, based on the buildings produced to navigate the tech tree. The resultning strategy tree is shown in Figure 1

The first leaf node that is reached determines the strategy label assigned to the player.

The rule set was then used for supervised learning of the strategies, resulting in a model that showed foresight of the strategies defined in the labelling

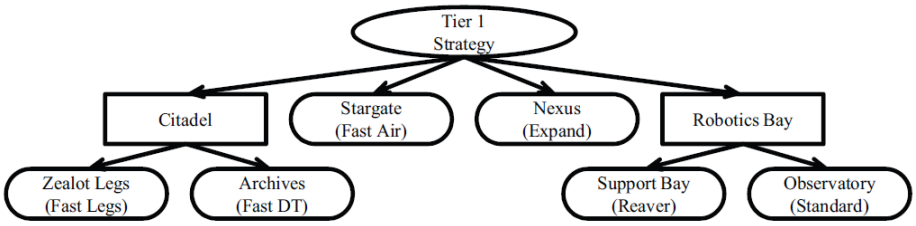

Figure 1: Rule set for labeling Protoss strategies. The tree is traversed by selecting the first building produced in the child nodes. Figure originally created by Weber et al[23]

tree. Furtheromre the robustness of the model when exposed to imperfect information was tested by presenting the model with missing information, and noisy information, both situations likely to occur in a game of SC.

In the article Weber and Mateas showed that data mining can be used to predict the label that will be assigned to a player in a replay, as well as showing that regression techniques can take advantage of the correlation between certain actions, such as the production of the observatory structure, and the production of the observer unit. The model is limitied by the rule set of the classifier, and thus the results of the prediction are only significant, if the classifier is designed to correctly describe significant elements of strategies.

2.3 PROBABILISTIC MODELS

Probabilistic models provides a convenient way to describe uncertainty, since we can describe relations between events with a degree of probability, and thus compute how likely a given chain of events is. In relation to strategy prediction and opponent modelling this allows for the computation of the probability of an opponent taking a specific action. This allows for an ability to encapsulate that certain plans may be more likely than others into the model[15]. This is very convenient when modelleing SC since both differences in player skill, and the risk of not observing a given feature, due to the incomplete information available to players, contribute to the uncertainty of prediction in the SC domain. On top of this, different game worlds may influence development, allowing for extensive variance across replays.

A useable probabilitstic model for plan recognition is the *Bayesian network*[9][10][11][12]. The domain can be modelled by variables, and a directed acyclic graph, which represents the probabilistic dependencies between the variables. Special kinds of Bayesian networks can be made if certain assumptions are made. *Dynamic Bayesian networks* for example, can be used to model systems that evolve over time. HMMs are a specialized version of dynamic Bayesian networks which allow for inference in polynimial time[14][20]. Synnaeve et al.[21] expands on the ideas that Weber et al. present[23] and focus on using these ideas for implementing an agent that can play SC. Synnaeve et al. expand on the ruleset made by Weber et al. to represent dependencies between opening stragies, ordering of building production, and the time played. This expanded model also takes into account the limited information

availability of SC since it includes a specific variable for observed buildings. Weber et al. define a player's strategy as the choice of units and buildings that the player constructs, with the intention of modelling and discovering strategies in SC. The game is divided into 30 second time steps and in each time step it is detected if a unit or building is constructed or not, that is as a boolean variable. In the article a strategy is described as the player's path through the state space of the HMM.

In the follow-up article[12], Dereszynski et al. try to make a more sophosticated model by incorporating the number of units that the opponent controls in each time step. This is achieved by incorporating the number of each unit type that is killed as well as the number that has been observed. This expanded model is no longer an HMM and the complexity of direct inference is no longer polynomial. The article circumvents this by using particle filtering and thus performing approximative inference.

3 THEORY OF HIDDEN MARKOV MODELS

3.1 THEORY OF HIDDEN MARKOV MODELS

An HMM is, as the name implies, a Markovian model of an evolving process, where certain elements are hidden, ie. not directly observable. This means that the only way to determine the state of the model is indirectly, through the observable elements of the model, the *emissions*. If we make an assumption that the process our system models has a chance of producing a given symbol for each observation type, depending on the state that the model is in, we can then make assumptions about the current state of the process, which in turn will allow for making inference about future states.

3.1.0.1 *Structure*

An HMM can be visualized as a series of hidden state nodes, each with a number of child nodes, each representing an observation that can be made in each interval. For each time interval a local model, called a time step is introduced. The probability set for a time step is disjoint from every other time step. However, the hidden state in a time step is connected to the hidden state of the next time step with a directed edge, called a *temporal link*.

To formally define an HMM the following elements are needed: The number of hidden states in the model, N, the number of states we can observe, M, The initial belief state of the model, A, the probability distribution of moving between states, B as well as the probabilities of observing a specific value, or symbol, in each state for each observation variable denoted as π. A, B and π are denoted collectively by λ. Thus an HMM is defined by the tuple (N, M, λ).

An HMM is in exactly one state at a point in time. This state is however not directly observable, only indrectly through the observations made in the time step. When moving between time steps the model can change its state. If the HMM is of the *ergodic* type, the new state can be the same as in the previous time step, whereas a model of the *forward* type does only allow for transitions to another state in subsequent time steps. For the purposes of this project, only ergodic models will be considered.

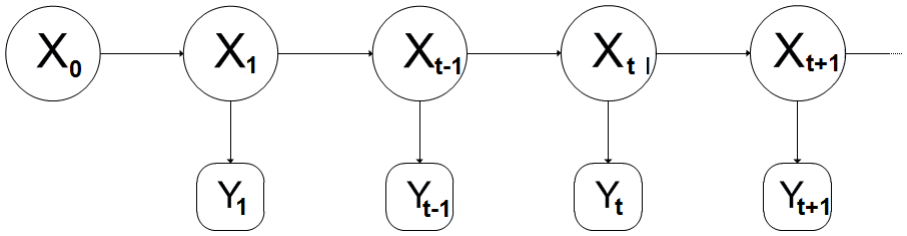

Figure 2: The general structure of a Hidden Markov Model

In Figure 2 the state nodes, labelled X can be seen connected by *diverging connections* to their child notes, the child nodes being the hidden state in the next time step X, as well as the observations for time step, labeled Y.

A diverging structure, as shown in Figure 3, means that the children, in the case of multiple children, in Figure 3 labelled as B, C and D, are individually independent, given the state of the parent node a. Thus, if we instatiate the parent node with a specific observation, no instantiation of a child node will influence the state probablities of the other child nodes. This is known as *d-separation*. If the parent node A is uninstantiated, however, information can flow through the parent and influence the state probability of the parent, as well as the other child nodes.

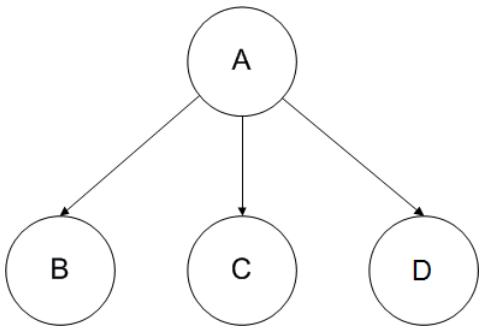

Figure 3: An example of a diverging connection. Picture originally shown in [14]

3.1.1 *Determining the Belief State of the Model*

Calculating the belief state $P(X_t|y_{1:T})$ in a time step t is a two-part process. The first part is to calculate the state probability distribution in the given time step based on the observations and belief states for previous time steps, as well as the observations for the time step in question. This process is called *filtering*. The second part is to refine the estimation by factoring in later developments by starting at the latest observation and working backwards to the relevant time step. This is called *smoothing*. Since the calculation fo the first step moves forward in time and the other backwards, they are known as *forward message propagation* and *backward message propagation*, in scientific literature denoted by α and β respectively.

Calculating the forward message

To calculate the forward message for a given time step, α_t, we need to know the belief state of the model in the previous time step, in order to know the transition probabilities to the current time step. This can be formally defined as:

$$\begin{aligned} \alpha_t &= P(X_t|y_{1:t}) \\ &= P(X_t|y_{1:t-1}, y_t) \\ &= P(y_t|X_t) \sum_{X_{t-1}} P(X_t|X_{t-1}, y_{1:t-1}) P(X_{t-1}|y_{1:t-1}) \end{aligned} \tag{1}$$

For the initial time step no prior belief state exists, so instead the *initial probability distribution* is used. In order to factor in the observed variables we take the emission probabilities of each variable observed in a given time step, and multiply the probabilities of the observations. This yields a distribution, $P(X_{t-1}|t_{1:t-1})$ to use for calculating the forward message for the next time step. This likelihood is normalized into a probability distribution which is used as the baseline for calculating the forward message for the next time step. This is then done for time *t=1* to *T*.

A full expansion of Equation 1 can be found in [7].

Calculating the backward message

The claculation of the backward message for a given time step, β_t, follows much the same procedure as the forward message, except that it works backwards from the last observed time step to the time step in question. This can be formally defined as:

$$\begin{aligned} P(Y_{t+1:T}|X_t) &= \\ &= \sum_{X_{t+1}} P(y_{t+1:T}, X_{t+1}|X_t) \\ &= \sum_{X_{t+1}} P(y_{t+1}|X_{t+1})P(y_{t+2:t}|X_{t+1})P(X_{t+1}|X_t) \end{aligned} \tag{2}$$

Since no data exists beyond time step T, we do not have a state belief beyond *T* and thus need an initializer to start the recursive backwards calculation. As we have no data, we must assume that all states are equally likely, so the initial value is set to 1 for all states. From here we proceed to multiply the emission probabilities of the observed variables per state in the previous time step, with the transition probability matrix. For the backward message it is worth observing that we cannot calculate a value for time step *t=1*, since there are no observations for *t-1*, as there is no previous time step.

Avoiding underflow

Since the calculation of the forward and backwards message involves multiplication of several variable of less than zero, a risk of underflowing exists. To alleviate this, *scaling* can be used. The scaling factor is the normalization factor of the α calculation for each time step. The scaling factor for a time step is denoted c_t. The β values for each time step is then divided by the scaling factor from the α calculation in the same time step. This ensures that the α and β calculations are weighed equally, even after the scaling.

3.1.2 *Learning*

An integral part of the use of HMMs is the ability to have a model learn an approximation of the parameters that govern a domain. Since the parameter space of an HMM is very large, it is infeasible to directly determine the variables directly. Instead an iterative refining must be used to update the parts of the model: The initial state distribution, the state transition model, and

the model emission probabilities. For the purposes of this report, it is worth noting that the learning will be performed using several game replays as sources of information and that the learning equations have been updated to reflect this.

3.1.2.1 *Updating the initial state distribution*

To calculate the updated initial state distribution, we calculate the expected total number of transitions from the initial state x_0 divided by the total number of transitions across all replays and time steps. This can be expressed mathematically as

$$\hat{P}(x_0) = \frac{\sum_{r=1}^{R} P(x_0|y_{t0:T})}{\sum_{r=1}^{R}\sum_{t=1}^{T} N} \tag{3}$$

$P(x_0|Y)$ can be calculated as follows:

$$P(x_0|Y) = \alpha_{t_0}\beta_{t_0}P(x_0)P(y_0|x_0) \tag{4}$$

$P(x_0)$, the initial state probabilities are multiplied by the emission probability of each observation variable, $P(y_{t0}|x_{t0})$, based on the observed value for the variable, as well as the α value for the first observed time step, and corresponding β value. This is now done for all replays, and the values are summed over all the replays. Finally we divide by the total number of transitions in all replays. This yields the updated initial transition probabilities.

3.1.2.2 *Updating the state transition model*

For the general case we do not know how many transitions are made from each state, so the denominator will have to be updated to calculate the probability of being in a state and transitioning to a new state. The full formula for the calculation is then:

$$\hat{P}(x_t|x_{t-1}) = \frac{\sum_{r=1}^{R}\sum_{t=1}^{T} P(x_{t-1}, x_t|y_t)}{\sum_{r=1}^{R}\sum_{t=1}^{T} P(x_t|y_t)} \tag{5}$$

We can calculate this, by using the following calculation to find the nominator

$$P(x_{t-1}, x_t|y_t) = P(y_t|x_t)P(x_t|x_{t-1})\alpha_{t-1}\beta_{t+1}$$

$P(x_t|y_t)$ can be read in the emission probability distribution for each variable in each time step. For the nominator, we incorporate $P(y_t|x_t)$, that is, the sensor model, since this helps define which state we are in, based on the observations before and after the current time step. Similarly, we incorporate the transition model, since the previous state influences the probability of the current state, based on what has been learned in the previous iterations. We multiply with α and β, since these give belief state information for time t based on the observations of the previous and future observations in the chain.

3.1.2.3 *Updating the emission probabilities*

When calculating the updated emission probabilities we need to update for each emission variable individually. For each variable the update calculation is:

$$\hat{P}(y_t|X_t) = \frac{\sum_{r=1}^{R}\sum_{t=1}^{T} P(X_t, y_t|Y)}{\sum_{r=1}^{R}\sum_{t=1}^{T} P(y_t|Y)} \tag{6}$$

For the calculation of the nominator it is worth noting that $P(X_t, y_t|Y)$ can be rewritten to $P(X_t|y_t, Y)P(y_t|Y)$ by using the Bayesian fundamental rule [14]. This is relevant since we can observe that a specific value has probability 1 if the observation is actually observed in the learning sample in time step t of replay r and else is 0 since it is either observed or not. $P(y_t|Y)$ can be calculated by multiplying $\alpha_t\beta_t$ since these give information about the belief state of the model, for time step t. $P(y_t|Y)$ is also the calculation for the denominator of the equation, so here the same calculation is made.

3.1.2.4 *Termination*

The learning algorithm can be terminated in one of two ways: After a specific number of iterations, or when a specific threshold for likelihood increase is reached. Likelihood is a mathematical definition of how well the trained model suits the data after an iteration of learning. It is defined as $P(Y|\lambda)$, meaning the probability of the learned model parameters, λ producing the observations Y, where Y is the data used for training the model. $P(Y|\lambda)$ can be calculated as follows:

$$P(Y|\lambda) = \sum_{X_T} P(y_{1:T}, X_T|\lambda) = \sum_{T} \alpha_T$$

This is the same as the unnormalized calculation for filtering, so to account for this, the equation is updated to account for the multiplication of the normalization factor for a given time step, c_t. The equation then becomes

$$P(O|\lambda) = \prod_{t=1}^{T} c_t \sum_{X_T} \alpha_T$$

The result of this equation will always be 1. However, the following holds:

$$\prod_{t=1}^{T} c_t \sum_{S_T} \alpha_T = 1$$

$$\Updownarrow$$

$$\sum_{S_t} \alpha_T = \frac{1}{\prod_{t=1}^{T} c_t}$$

$$\Updownarrow$$

$$P(o_{1:t}|\lambda) = \frac{1}{\prod_{t=1}^{T} c_t}$$

Since the calculation involves multiplication of several values smaller than 1, we again risk underflow. To compensate for this, the $\log_2$ value can instead be applied to both sides of the equation. For logarithmic values, addition is the equivalent of multiplying[18] for a final result of

$$\log_2(P(Y|\lambda)) = \log_2 \frac{1}{\prod_{t=1} Tc_t} = \sum_{t=1}^{T} \log_2(c_t)$$

This calculation is performed for all R replays, for a final result of

$$\sum_{r=1}^{R} \sum_{t=1}^{T} \log_2(c_t) \tag{7}$$

Termination can now be decided on the basis of the increase in loglikelihood since the last iteration.

3.2 POISSON HIDDEN MARKOV MODELS

Poisson Distributions

A Poisson distribution is defined as [17]:

$$P(y, \lambda) = \frac{e^{-\lambda}\lambda^y}{y!} \tag{8}$$

where λ is the expected value of the observation, y is the actual value observed[1], and e is *Euler's number*. The result of the equation is the probability of observing y given the expected value, λ.

Poisson distributions that describe expected occurrences over time are found in real life in, for example, birth and death statistics, or the decay of nuclear materials, where we have a statistical average we can expect over a time period.

3.2.1 *Update rules for Poisson Markov Models*

Since all other elements than the emission variables of a PHMM are the same as those described in 3.1.0.1, the only new rule that needs be introduced is an update rule for calculating an update of the λ value of the Poisson variables. In each iteration, k, each Poisson emission variable will have a different λ value that will need to be updated for each state, i, that is, λ_i^{k+1}. This calculation is given by [17]

$$\hat{\lambda}_i^{k+1} = \frac{\sum_{r=1}^{R}\sum_{t=1}^{T}\alpha_t^k(i)\beta_t^k(i)y_t}{\sum_{r=1}^{R}\sum_{t=1}^{T}\alpha_t^k(i)\beta_t^k(i)} \tag{9}$$

Calculation of the forward and backward messages for a PHMM is done in exactly the same way as for a regular HMM, as described in Section 3.1.1 and Section 3.1.1, except that the emission probabilities are not read from an emission probability table, but calculated on demand, using Equation 8. Likewise, the same rules for termination can be used as for a regular HMM, with the same reasoning as presented in Section 3.1.2.4.

3.3 POISSON AND MULTINOMIAL MIXED HIDDEN MARKOV MODEL

Some emissions in a model may not be expected to occur repeatedly, while some others may. In the SC domain, non-repeating emissions would most likely be construction of base-structures. These will most often be built in specific numbers, at specific points in the game, and then no further observations are likely to happen, since these structures only serve two purposes: To produce combat or resourcing units, or to advance the tech-tree. For an explanation of the tech-tree concept, refer to 1.1.1.1.

1 not to be mistaken for the λ describing model parameters in HMM architecture, which unfortunately uses the same symbol

Repeating emissions will be production of units , where units are constructed to either replace dead units or reinforce the number. Unit deaths are assumed to follow a similar pattern of repeating occurrence.

To properly model this, a mixture of regular HMMs and PHMMs is needed. For this no new rules are necessary, since the update rules for X_0 and γ are the same for both types of models. For the emission updates, we only need to know the type of emission, and apply the relevant update equation to the emission

4 DATA ANALYSIS

4.1 DATA AVAILABILITY

SC has a feature of players being able to save replays of multi-player matches. The replay is saved as a file that can later be played back, much in the same way as video. These files are small and thus easy to share, leading to the rise of several websites where collections of replays are available for browsing. This makes SC a reasonable source of data for Machine Intelligence, but with certain reservations: SC matches are deterministic, meaning that if the same set of actions is made at the same time, two matches will play out in the same way. This means that the replay files, will only have to record the faction of each player, the orders that were given, and the time stamp of the order. This accounts for the small size of the replay files, but present data miners with a problem: The replay does not contain any information about unit deaths, or destruction of buildings, which in turn means that the current existing number of a given unit or building type is unknown, unless the replay is played back, since only the construction orders are present in the replay file. A workaround for this needs to be found for the full information potential of a replay to be utilized. The process of extracting data from SC replays involves finding software to log the events in the replay as they happen, as well as converting these log-files into features that can be used in an HMM. This process is described in Chapter 5.

The replay pool has been acquired by crawling the SC competitive communities *GOSU gamers*[3] and *teamliquid.net*[6] and downloading the one versus one games with the protoss versus terran racial setup, for a final replay pool of 1800 replays.

4.1.1 *Data quality*

Despite the relative ease of acquiring SC replays, it is less easy to ensure the quality of the replays, that is, how representative a given replay is of an average SC game, since SC allows players to create custom maps – maps with special rule-sets defined by the user, which may use a distorted version of the standard rules, or even a different set entirely. Since it is infeasible to manually investigate every replay, certain criteria have been applied in the attempt to clean the dataset: Replays that displayed production values of more than seven of the same unit in a time step were filtered, since it was deemed unrealistic to reach this level of production within the seven minute threshold that is analyzed. Similarly, replays where either player did not produce any worker units were removed as well. This reduced the pool of learning data from 1800 replays to 1400. In the same vein as the problem of custom maps, is the skill level of the players involved. A skilled player may

not only be much faster than an amateur, but may also employ strategies that amateurs may not know, or be able to perform, due to the required skill level of the strategy. Optimally, replays should be obtained from the games played in the professional leagues, but this is unfortunately not probable, since the replays of active players are similar to recordings of other sporting events: guarded by investors who want to make money, and by players who don't want their opponents to study their strategies. This leaves the replays of inactive players, which, unfortunately quickly get lost to time.

When deciding which features to model as Poisson distributions, and which to model as multinomial, it is important to analyze the patterns exhibited by the variables in the data.

4.1.2 *Data patterns*

Intuitively, one could expect that the production and destruction of units would follow the repeating pattern of Poisson distributions, since the units are the effect of a specific strategy. Construction of buildings would be more sporadic, as these would be more indicative of the enabling of a strategy, by traversing the tech tree, and production of the units that are part of the strategy, and would therefore only occur in brief spurts. If this intuition holds true, the production of a unit would be expected to show a tendency of several units produced in a single time step of 30 seconds, and construction of buildings should rarely exhibiting values of more than one in a 30 second interval.

4.1.3 *Production*

For the best overview of the production, it should be investigated how frequent a given production amount is for the type, as well as the total number built in the replay. In combination, these will give an overview of not only if the construction of a given unit or building is a repeating process, but also how many can be expected to be seen once production starts.

As a sample, production of the *zealot* and *dragoon* units have been selected to be shown, and the *gateway* and *cybernetics core* for building construction. These have been selected since the two units are the earliest accessible combat units in the tech-tree and thus have the longest time frame where they can be expected to be observed. The *gateway* is the structure that produces both units, with the *cybernetics core* being required for production of *dragoons*. The production of the *observatory* structure is analyzed as well, since this is one of the possible progressions from the production of the units enabled by the gateway structure. As an example of a more advanced unit, the *dark templar* has been chosen since this unit is also built from the gateway. The statistics have been made using all the replays used to train the models. The first time step shown in each model, is the first time step in which the event is observed.

Figure 4 shows the number of replays where a specific number of zealots existed in the given time step. From this we can see that zealot production

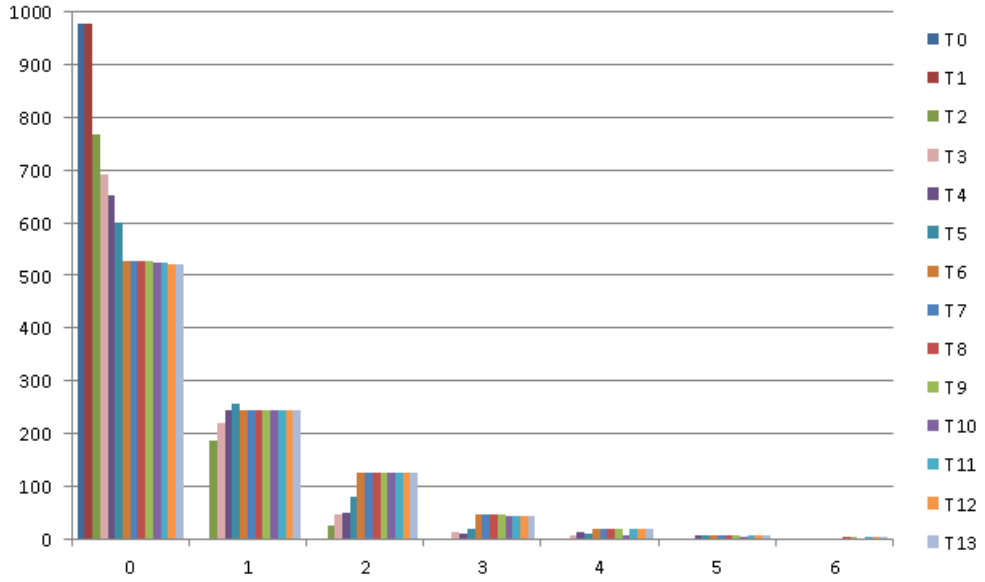

Figure 4: The total number of zealots produced in a replay is the x-axis, and number of replays is y

follows a pattern where only one or two are produced during the first seven minutes. If we cross reference this information with the graphs in Figure 5 we can make an assumption that these zealots will be produced in the state that the learned model assumes around the 150 second mark. This one-time production makes sense in the context of SC since the racial setup of the data is a *Protoss* player vs. a *Terran* player, and the basic combat unit of the Terran race, a *marine*, is more cost-effective against zealots than the other way around. The Protoss player will thus seek to progress to another unit as soon as possible, while still having a small force of zealots to protect his base against early raids from the enemy. It does, however, raise questions regarding the validity of modelling zealots as a Poisson distribution, since zealot production seems to be a one time event. It seems probable that learned models will thus only have one or two states that produce zealots, and no more than one.

Looking at Figure 6 we can see that the production number of gateways is approximately 1.5 on average. If we cross reference this with Figure 7 it seems to corroborate the assumption that production of gateways happens once, in time step T2 and a secondary gateway for expanded production capacity, is then constructed later. This insinuates that gateway production is a discrete event and thus not suitable for representation as a Poisson distribution. This also makes sense in the context of SC where the first gateway will be built as fast as possible to allow for production of units to fend off potential early raids, with resources allocated later to allow for expanded production, after the player's position has been reinforced. For these reasons, the multinomial distribution seems more appropriate, since the focus then is more on *when* the production happens, than how many are produced.

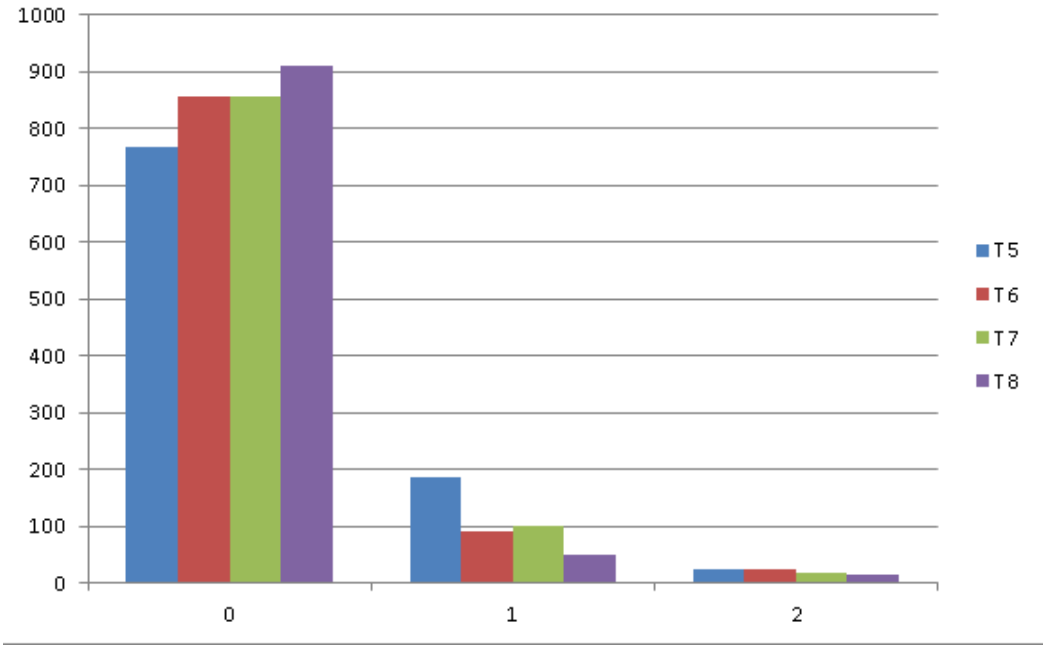

Figure 5: Total number of zealots produced up to the time step. X is the produced number, Y is the number of replays

If we look at Figure 8 and Figure 9 we can see that the cybernetics core follows a similar pattern to the gateway – that construction is a singular, discrete event. This makes sense, since the cybernetics core serves the following purposes: Unlocking production of Dragoons, opening for the construction of the *Stargate* structure, and upgrading the flying units produced from the Stargate. Producing more than one cybernetics core would then make little sense, since the marine unit can attack air units as well as ground units, and thus there is little point in building a Stargate, and much less in upgrading the air units it produces, until an efficient counter to the marines has been produced.

Figure 10 shows the frequency of the total number of produced dragoon units. The dragoon is the first combat unit unlocked after the zealot in the tech tree, enabled by the construction of the cybernetics core. The dragoon is more effective against the marine unit than zealots, as well as being a reasonable response against the next units in the traditional Terran progression: The fast moving *vulture* raiding unit or the *siege tank*. For this reason it is reasonable to expect a higher production of dragoons that is more consistent across time, which is also what we can see from Figure 11. This could indicate that a Poisson distribution is good for describing the production behaviour of dragoons, as this will allow a finer granularity of the expected number of dragoons produced in a time step, than a multinomial distribution.

Looking at Figure 12 we can see that there is never built more than one observatory structure in the first seven minutes of a game. This makes sense, since the observatory structure only serves to enable production of the *observer* unit, much in the same way as is the nature of the dragoon / cyber-

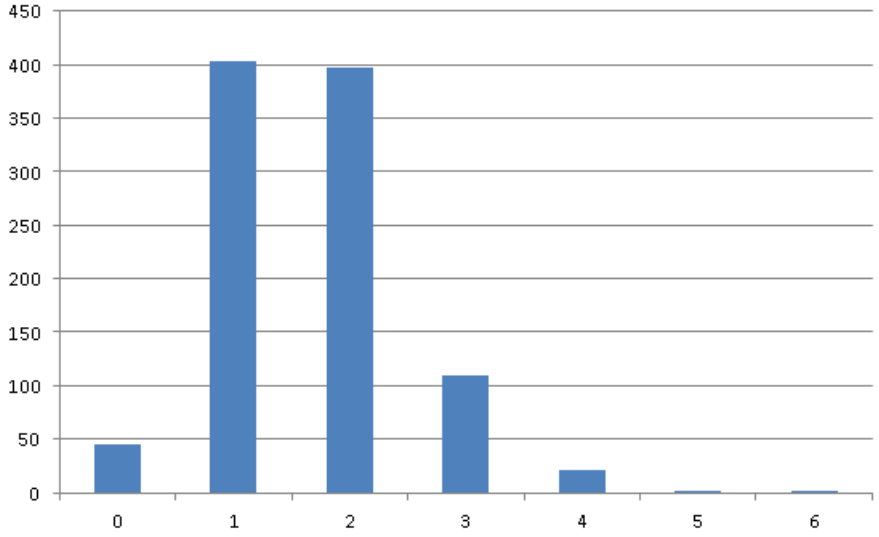

Figure 6: The total number of gateway structures produced in a replay is the x-axis, and number of replays is y

netics core relationship, and to make upgrades for the observer units. We can also see that if an observatory has not been built by time step T6 it is not going to be built. Thus the argument becomes much the same as for the cybernetics core structure – that the interesting question is how likely the construction is to happen, and not how many are produced.

If we look at Figure 13 we can see a similar pattern as that for the zealot unit in Figure 5 – that the production is in fact a one time event. If production of dark templars has not begun by time step T6 it is not going to happen. This does make sense in relation to SC since the dark templar is a unit which is effective in specialized circumstances only. As is the case for the zealot production, this makes the decision to make all units be represented with Poisson distributions less ideal.

4.1.4 *Destruction*

As important for the measurement of the opponent's army strength, the pattern of production is, just as important is the pattern of destruction. Intuitively, one would expect to see a few repeated losses in the early parts of the game, as small skirmishes are fought, and, later in the game, shorter periods of large repeated losses, as the players commit their forces to gain the upper hand, and send produced reinforcements to the front lines.

Figure 14 shows the number of dragoon units destroyed in the time step. The figure seems to follow the pattern of small skirmishes – some units are destroyed from T6 and onwards, and in sligthly increasing numbers. This means that we can expect the model to show a λ value for destruction, to be somewhat close to 1 from T6 and onwards. If we combine these two observations, we would expect the number of live dragoons to be increasing

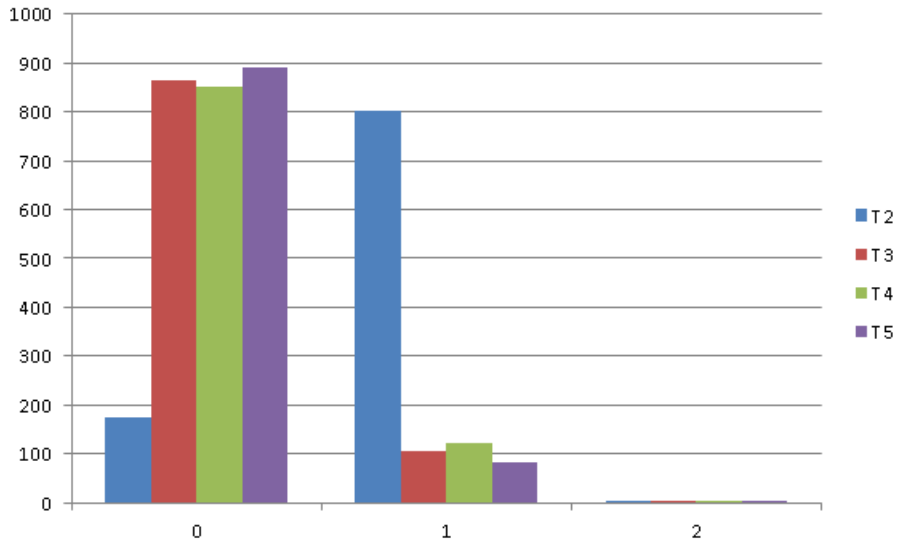

Figure 7: Frequency of replays where the number of gateway structures were produced in the time step

at a rate of approximately half a dragoon per time step after production begins in time step T6.

4.2 DATA CONCLUSIONS

Based on the the analysis of the data, it seems to be a good idea to keep the construction of buildings as a multinomial distribution, whereas the switch for units to Poisson distributions is a bit more ambigous. This ambiguity is enhanced by the relatively short time window that is analyzed. Production of more advanced units will be in the early upstart phase, if being produced at all, which makes it difficult to analyze the validity of assuming that they are continous productions. In addition to this, certain units may only be cost effective in a certain time frame, that is, until the opponent is able to adapt his unit production to something that effectively counters this unit, the dark templar being an example of this. If production of dragoons is started however, the production seems to follow a repeating pattern from time step T6 and onwards.

This does not, all in all, make for a convincing case for switching all unit representations from multinomial to Poisson distributions, but assumptions are not always backed up by experiments. A better case could perhaps be made for the switch if the model was made for longer time periods of SC games, such that production was able to get up to full speed, and the more advanced unit was given time to be produced.

4.3 INCREASED TIMESPAN

To investigate if the investigated time period is the determining factor in the patterns exhibited by unit production, more time steps have been investi-

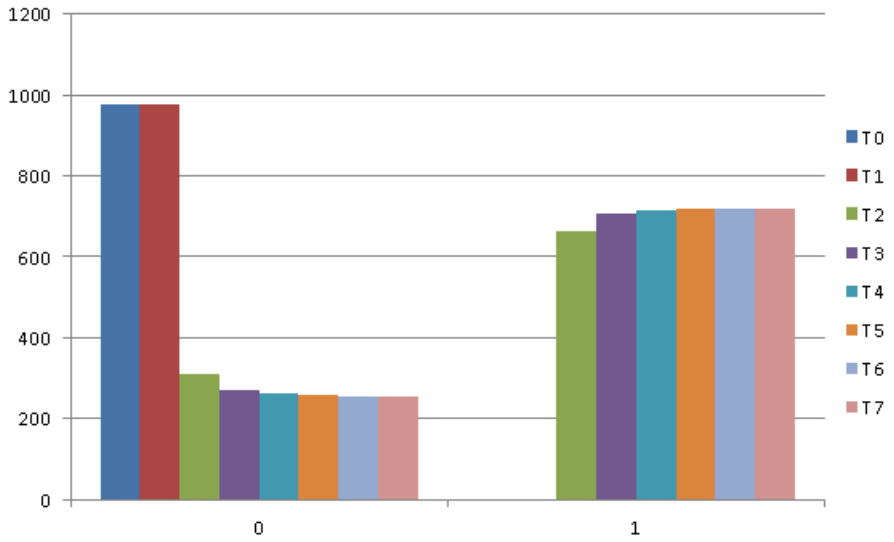

Figure 8: The total number of Cybernetics Core structures produced in a replay is the x-axis, and number of replays is y

gated. The production of zealot units and dragoon units in a 10 minute time frame has been investigated.

Figure 15 shows that after the decline in production from timestep T6, production resumes again in T12 to T14 and remains fairly consistent from there. This makes sense in an SC context, as the terran player will usually try to reinforce his position by producing more siege tanks, which zealots are useful against. If this thesis holds, we should expect that production of a unit that effectively counters marine production should commence, to allow the zealot units to fight siege tank units instead of marines. One such unit would the *reaver*. If we look at Figure 17 and Figure 18 we can see that production of reavers not only starts in time step T8, but that the number of of reavers is almost completely static, indicating that the production in later time steps is to replace destroyed reavers. Adding to the inference of this strategy, is the escalation of a continued production of dragoons, as can be seen in Figure 16, which insinuates that the primary strategy of protoss players against terran players is to produce a force to protect against early raids, start producing dragoons to counteract the terran player producing siege tanks, and finally, starting the production of reavers to make the terran player's marines ineffective.

4.3.1 *Conclusionss on the extended timespan*

This expanded time frame indicates that if the model is intended to describe the overall game, instead of only the opening strategies, Poisson observations for unit production seem much more appropriate. For the same reason, models trained in Chapter 5 will be trained to learn 10 minutes of games, instead of only 7, as was the case in the previous project[7] as well as the article the previous project was based on[11].

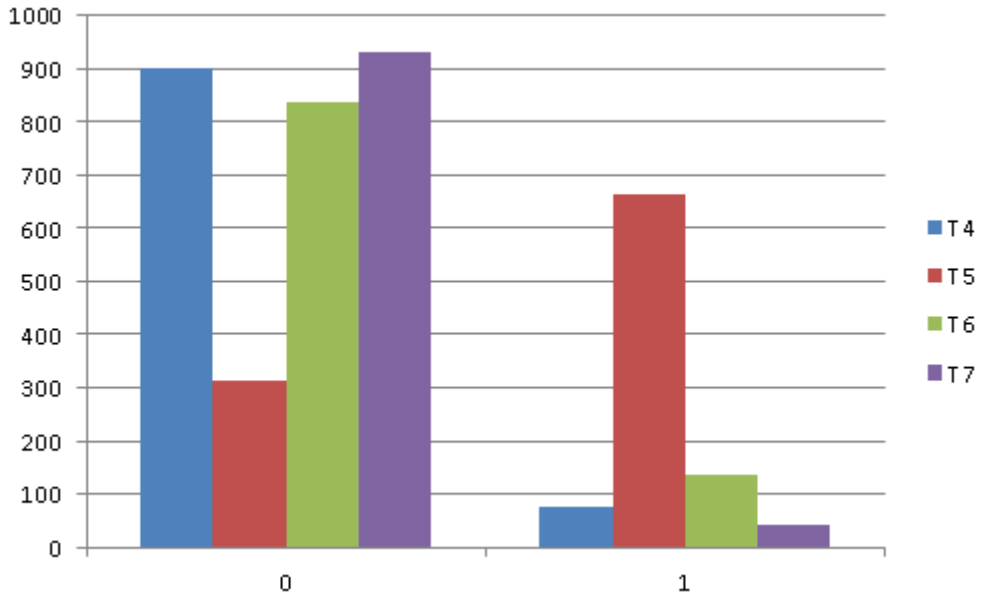

Figure 9: Frequency of replays where the number of Cybernetics Core structures were produced in the time step

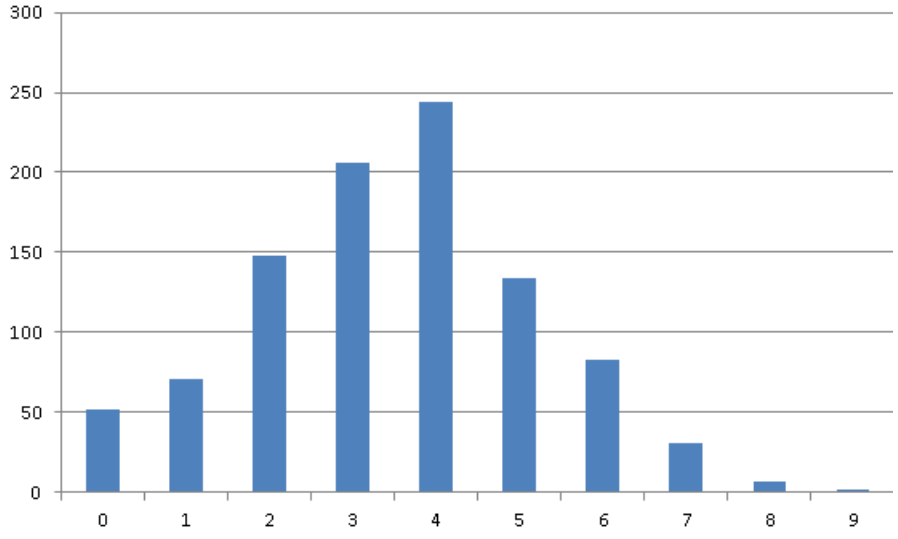

Figure 10: The total number of dragoons produced up to the time step is the x-axis, and number of replays is y

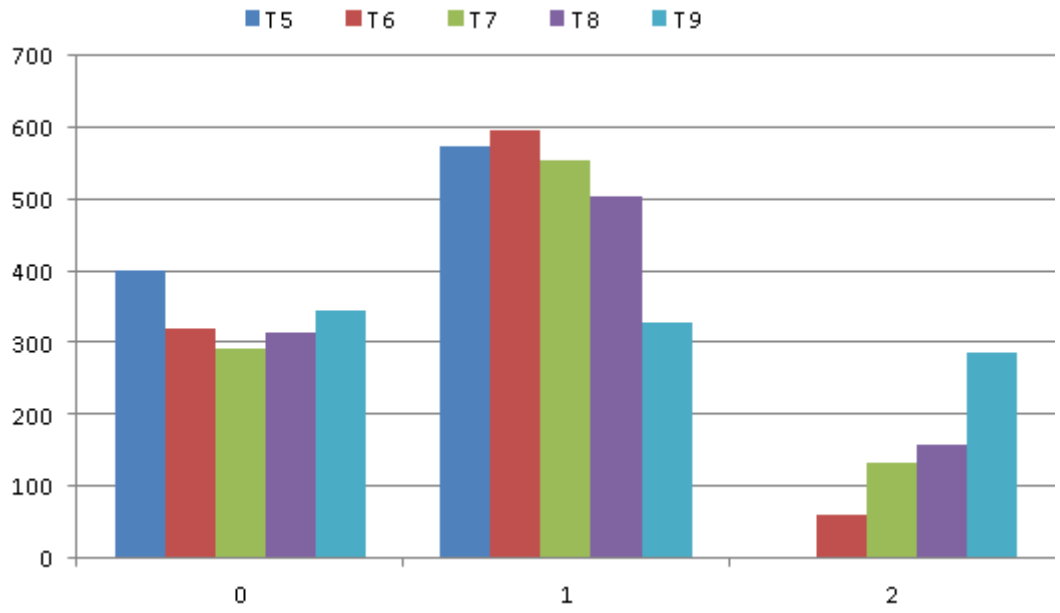

Figure 11: Frequency of replays where the number of dragoons were produced in the time step

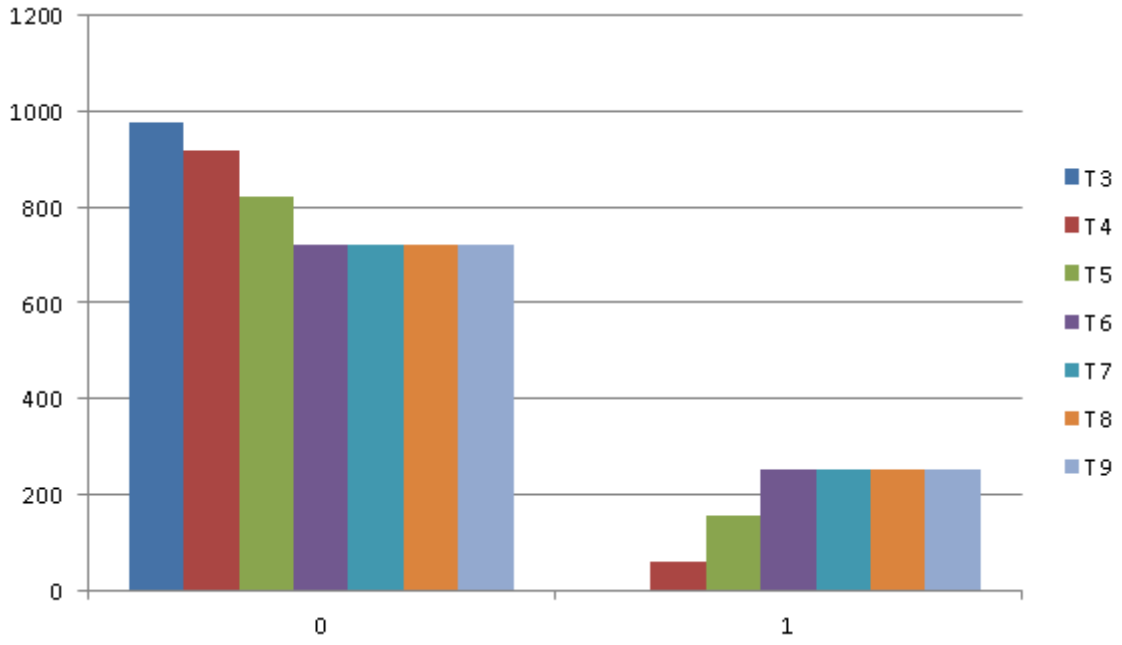

Figure 12: The total number of observatory structures in existence in the time step

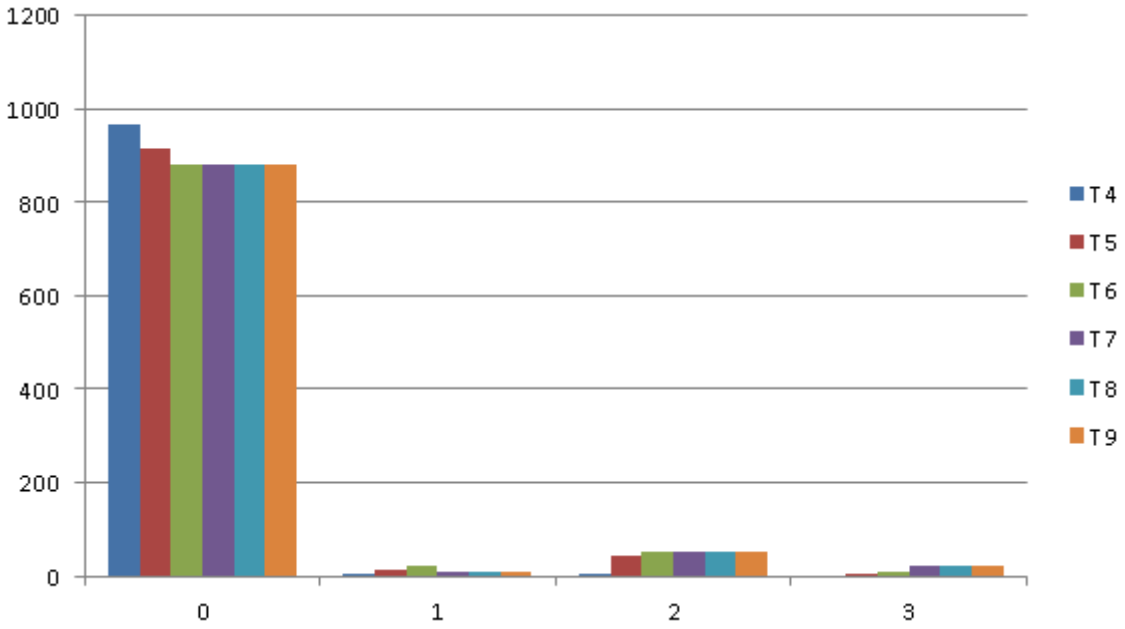

Figure 13: The total number of dark templar units produced up to the time step

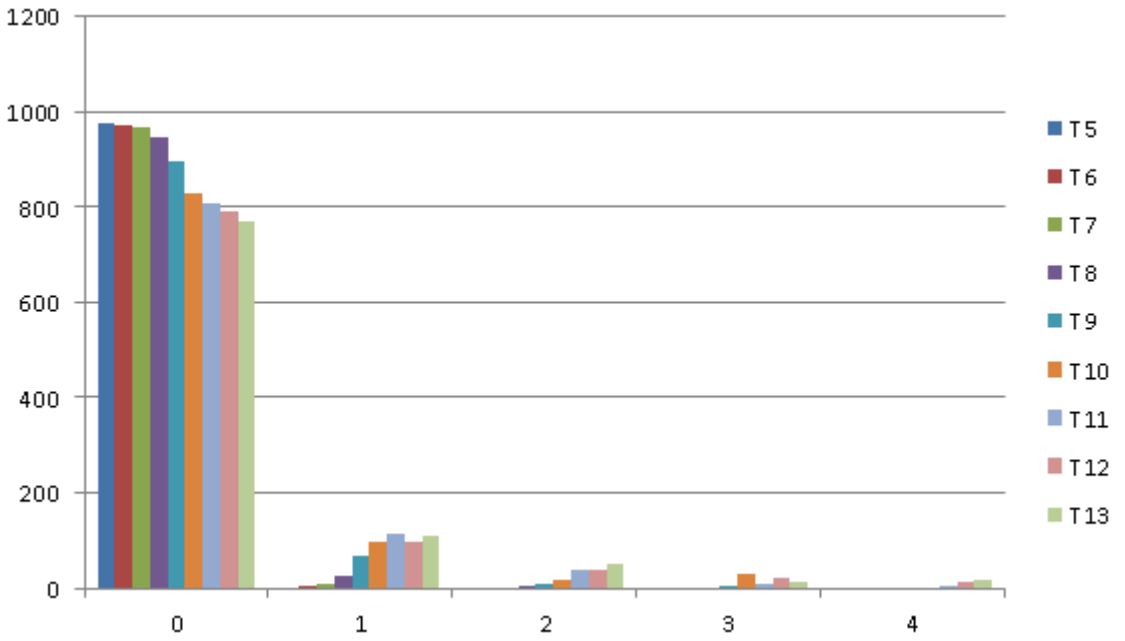

Figure 14: The number of dragoons destroyed in the time step

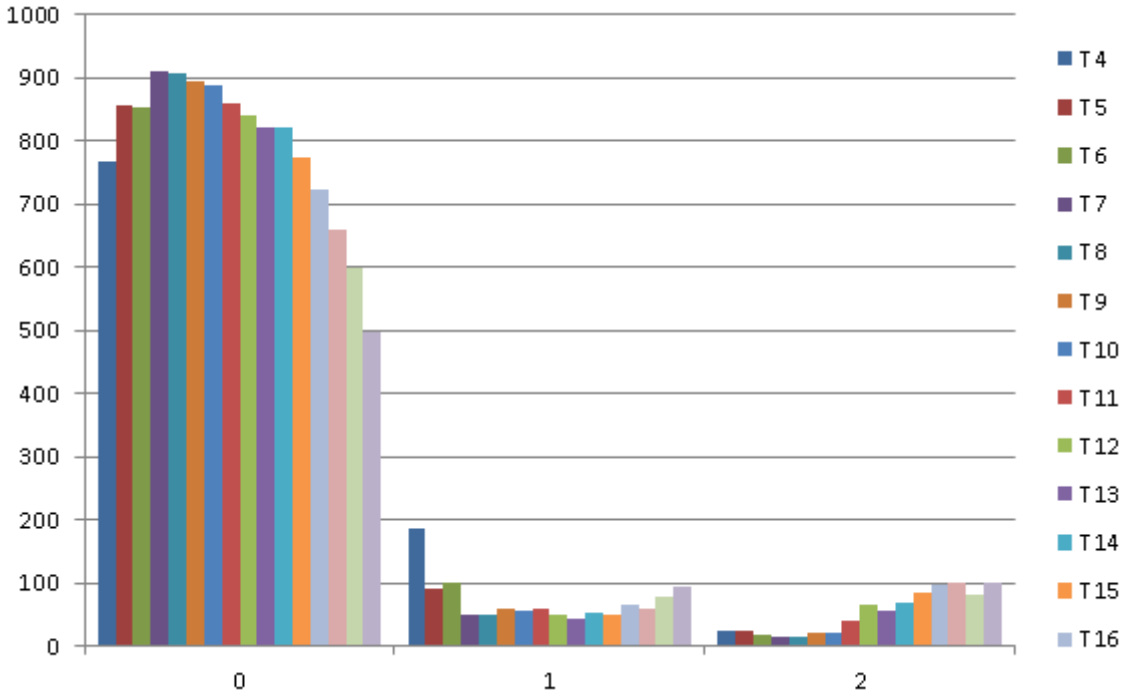

Figure 15: The number of zealots produced in the time step

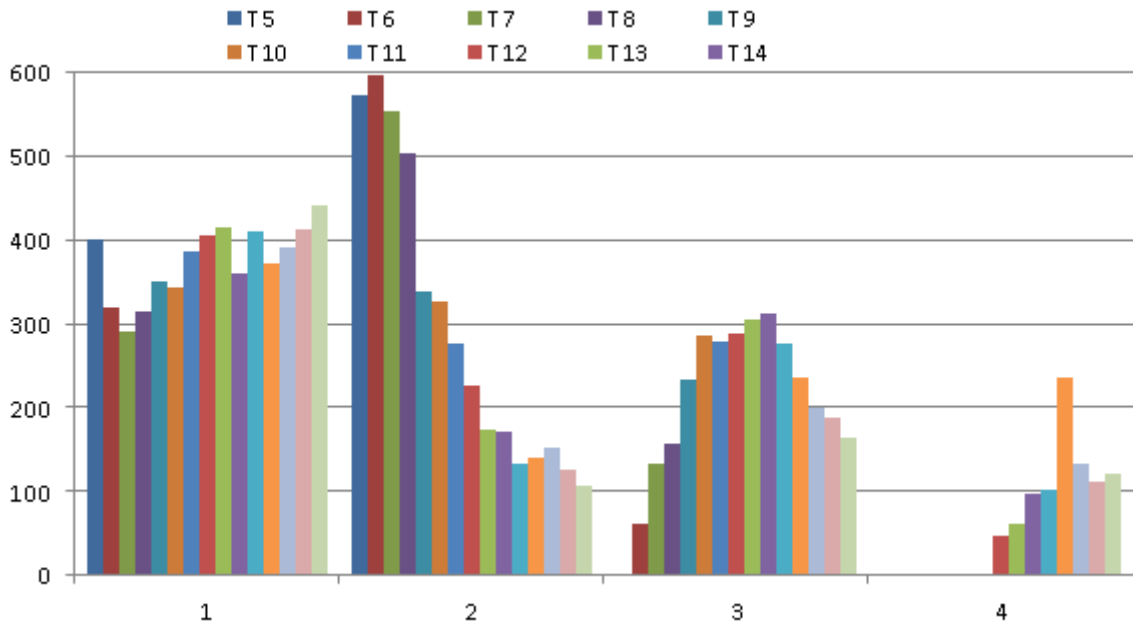

Figure 16: The number of dragoons produced in the time step

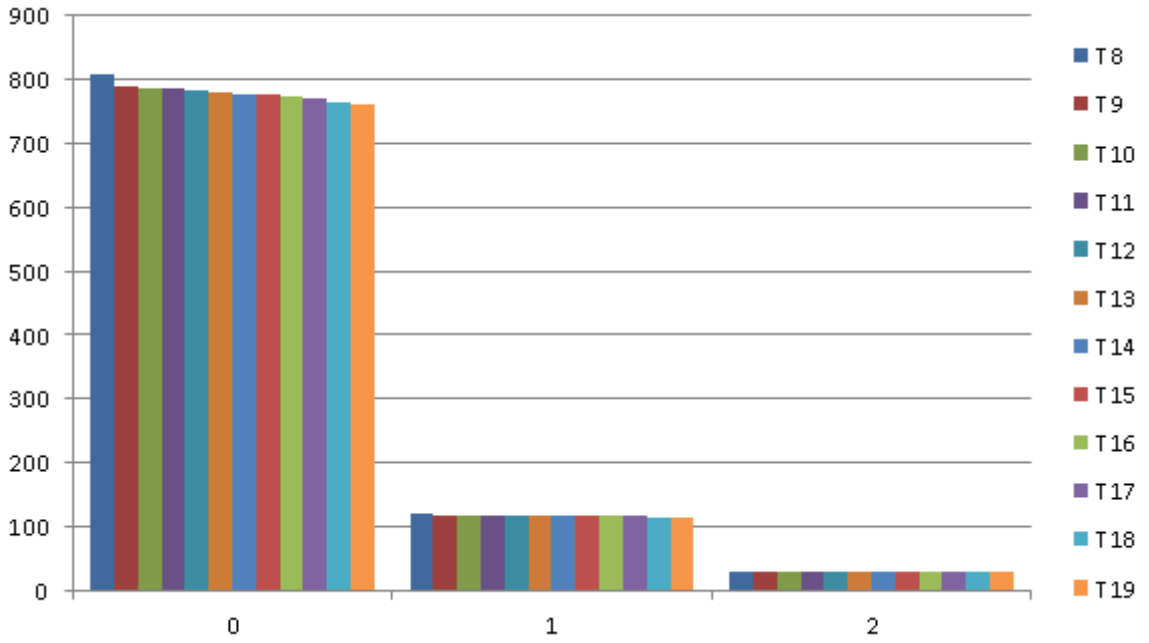

Figure 17: The number of reavers alive in the time step

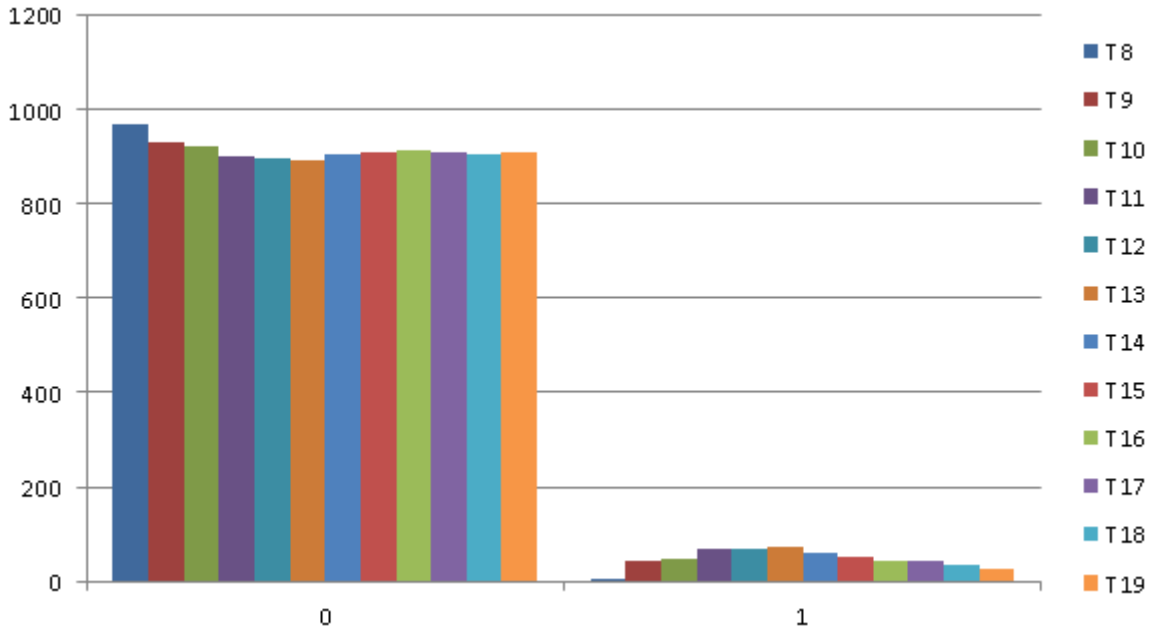

Figure 18: The number of reavers produced in the time step

5 IMPLEMENTATION

Several steps need to be taken to get from a collection of SC replays to a PHMM. First, a way to extract the data from the replays must be found, then this has to be parsed into something that can be used as input in a PHMM, and a model that can work with both multinomial and Poisson variables then has to be constructed. An implementation of a standard HMM written in C# from the *Accord.NET framework*[1] was originally intended as a baseline for creating a mixed type model, but due to errors found in the framework implementation along the way, the entire code was eventually rewritten from the bottom up.

5.1 REPLAY DATA EXTRACTION

As was mentioned in Chapter 4, SC replays are somewhat limited in the data contained directly in them, due to the deterministic nature of SC. Unit destruction is not present in the replay itself, since this will be implicitly contained in the game rules, based on the orders given by the players during the replay. This means that, in order to extract this data, each replay must be simulated, and the time stamp of the production and destruction events can then be recorded from the event log. Accessing the event log is not entirely trivial however. This has to be done through a third party application such as Brood War API (BWAPI). BWAPI is a code injection API that allows for both controlling the game and reading the game state through C++ programs[2].

Once BWAPI has been configured into a working configuration, logging the data is rather simple, as is demonstrated by the code in Listing 1.

The loop in line 1 of Listing 1, the loop runs while a game is in progress. First, the game client is updated, that is - it updates the frame, all the actions, the event list and the UI. The event list is now iterated across and switched for the type of the event, in this case if the event type is *UnitCreate* for line 11 or *UnitDestroy* for line 15. If an event falls into either case, the event is written to the log file through the logWriter stream writer with the type of unit, the type of event, and the number of frames into the game that the event occurred. Even though the events are named *UnitCreate* and *UnitDestroy*, these events apply to buildings as well.

5.1.1 *Data parsing*

The output from the BWAPI replay ripper is a log-file of all the production and destruction events in each replay. This needs to be parsed into something that can be used as input into a HMM, so a simple parser was written in C# to get from Listing2 to Listing3. This parser was given an ordered list of the feature names. Each replay log was then read line by line, splitting each line

```
1  while(Broodwar->isInGame())
2  {
3          BWAPI::BWAPIClient.update();
4
5          for(std::list<Event>::const_iterator e = Broodwar->getEvents().
               begin(); e != Broodwar->getEvents().end(); ++e)
6          {
7                  switch(e->getType())
8                  {
9
10                 case EventType::UnitCreate:
11                         logWriter << e->getUnit()->getType() << ",
                               created," << Broodwar->getFrameCount() <<
                               std::endl;
12                         break;
13
14                 case EventType::UnitDestroy:
15                         logWriter << e->getUnit()->getType().c_str() <<
                               ",destroyed," << Broodwar->getFrameCount()
                               << std::endl;
16                         break;
17                 }
18         }
19 }
```

Listing 1: The logging code for a replay, using BWAPI

into a string array based on the commas, such that the each lines was tuple of [Unitname][Eventtype][Timestamp].

Each tuple in the replay was then added to the relevant feature index in the feature vector, unless the time stamp indicated a new feature vector should be created. The feature vectors were decided to be 15 second intervals, since this was deemed the smallest meaningful time step in the SC domain. The time stamps were assumed to be 24 frames per second[4], meaning that each 15 second time step is 360 frames. This is the basis for Line 7 in Listing 4. These feature vectors can then be collapsed if longer time steps are desired, such as the 30 second time steps used in this project.

Listing 4 show the first part of the code that recognizes Protoss actions and indexes them in the feature vector according to the name. In this case, only the section for creation of new units or structures is shown, since the recognizer for destruction of units functions in much the same way. The method in Line 14 called *GetFeatureNumberFromName* does what the name implies: gives the index in the feature vector of the given observation, based on the name. Once all lines in the replay have been parsed, the result is saved as a .txt file.

```
Terran_SCV,created,21
Protoss_Probe,created,21
Protoss_Probe,created,416
Terran_SCV,created,431
Protoss_Probe,created,718
Terran_SCV,created,733
Protoss_Probe,created,1020
Protoss_Pylon,created,1223
Terran_SCV,created,1337
Protoss_Probe,created,1386
Terran_Supply_Depot,created,1469
Terran_SCV,created,1639
Protoss_Probe,created,1744
Terran,minerals,32,1800
```

Listing 2: Example of log contents before parsing

```
[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,0,0,0]
[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0]
[1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0]
[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0]
```

Listing 3: Example of feature vector after parsing. This example has been shortened – the actual vectors have 60 elements

```
if (Regex.IsMatch(elements[0].ToLower(), "protoss_")
{
        string unitType = elements[0].Substring(elements[0].IndexOf("_")
             + 1).ToString();

        if (elements[1].ToLower() == "created")
        {
                int timeSlizeNo = Convert.ToInt32(elements[2]) / 360;
                if (timeSlizeNo > featureVectors.Count - 1)
                {
                        while (timeSlizeNo > featureVectors.Count - 1)
                        {
                                featureVectors.Add(new List<int>(new int
                                    [featureNames.Count]));
                        }
                        featureVectors[timeSlizeNo][
                            GetFeatureNumberFromName(unitType.ToLower())
                            ]++;
                }

                else
                {
                        featureVectors[timeSlizeNo][
                            GetFeatureNumberFromName(unitType.ToLower())
                            ]++;
                }
        }
```

Listing 4: The code that recognizes Protoss actions and indexes them

```
1 PoissonMarkovModel myModel = new PoissonMarkovModel(3, 10, 32, 28);
```

Listing 5: invocation of the constructor for a mixed observation HMM

5.2 MIXED VARIABLE TYPE HIDDEN MARKOV MODEL

As was mentioned in the start of the chapter, the model was implemented in C#. The model consists of the following elements:

- A quadratic 2D array to represent the transition matrix. Each dimension size is the number of states. Each column denotes the previous state of the model, and the row the current state, such that $P(x_t = [x_t, x_{t-1}$
- A list of emission probability matrices for the multinomial distributions, represented as 2D arrays. Each array is ordered such that rows are the state number, and columns are the value of the emission indexed directly. Based on the data analysis, the values of the multinomial distributions are $0, 1, 2$ since multinomial values are used for production of buildings.
- An array of initial state probabilities.
- A list of the λ values for each Poisson observation in each state. For each Poisson variable is an array that contains the lambda value for the emission, given the state.

When creating an HMM through the constructor, one must define the number of values that the multinomial variables can assume, the number of states, the number of multinomial observation and the number of Poisson observations. A sample of this can be seen in Listing 5 where a model is created with 10 states, 32 multinomial observations, that can assume the values $0, 1, 2$ and 28 Poisson observation variables.

When a new model is instantiated, all elements of the model are assigned random values. The 2D arrays for transitions and multinomial emissions are then normalised by column, and the initial probabilities are normalised as well. The random instantiation helps ensure that the optimisation surface is explored, since complex problems have very complex optimisation surfaces with several local maxima[18].

5.3 FORWARD CALCULATION FOR MIXED VARIABLE MODEL

The calculation for the forward message shown in Equation 1 defines that, in order to calculate the forward message for a time step we must calculate the probability of being in each state, based on the the transition probabilities from the previous time step, and the emission probabilities for the current time step. To do this, not only for multiple observation variables, but for different types as well, proper circumspection is due. First of all, a way to incorporate both multinomial and Poisson variables must be made, and a way

```
1  //for timeslize 1 to T
2  for (int t = 1; t < T; t++)
3  {
4          //for each state s_t
5          for (int s_t = 0; s_t < States; s_t++)
6          {
7                  //probability 'prob' of being in state i and observing
                       sequence y(t)
8                  double prob = GetEmissionProbability(s_t, observations[t
                       ]) * calculateProbabilityForPoisson(s_t,
                       observations[t]);
9
10                 double sum = 0.0;
11
12                 //for each state j,
13                 for (int s_previous_t = 0; s_previous_t < States;
                       s_previous_t++)
14                 {
15                         sum += fwd[t - 1, s_previous_t] *
                               transitionMatrix[s_t, s_previous_t];
16                 }
17                 fwd[t, s_t] = sum * prob;
18
19                 normalization[t] += fwd[t, s_t]; // normalizing
                       coefficient
20         }
```

Listing 6: The induction step of the forward calculation

to differentiate between the types of observations. The easiest way to differentiate between types is to simply have all of the same type listed together, and then the other type. In this implementation, all multinomial variables are first, and then all the Poisson variables. If we refer to Listing 5 we thus have multinomial observations for the first 32 features, and Poisson observations for the next 28, for a total of 60 observation variables in each feature vector.

When calculating the forward message, the result is stored in a 2D array with each row being the time step, and the columns being the state. Thus, when referring to Listing 6 we can see in line 2 that we iterate over each time step in the observation sequence. In each time step we calculate the probability of being in state s_t by first calculating the probability of being in state s_t based on the observed variables in line 7 by multiplying the probability of the multinomial observations given the state, with the probability of the Poisson observations, given the state. In line 14 we then calculate the probability of transitioning to state s_t given the belief state of the previous time step and this is multiplied for a final probability of state s_t in time step t.

The two methods, *GetEmissionProbability* and *calculateProbabilityForPoisson* in line 8 of Listing 6 calculate the probability of a state, given the observation sequence. *GetEmissionProbability* simply looks through the emission probability tables for the multinomial observations, the number of which

```
1  private double calculateProbabilityForPoisson(int state, int[]
       observations)
2  {
3          double result = 1;
4          int observationOffset = observations.Length - poissonVariables;
5
6          for (int poissonParameterNo = 0; poissonParameterNo <
               poissonVariables; poissonParameterNo++)
7          {
8                  result *= calculateProbabilityForPoisson(
                       poissonParameters[poissonParameterNo][state],
                       observations[observationOffset + poissonParameterNo
                       ]);
9          }
10
11         return result;
12 }
```

Listing 7: The method for calculating the probability of a state, given an observation sequence

were defined in the constructor, and multiplies the emission probabilities together. The function for *calculateProbabilityForPoisson* is more interesting since the probabilities of a Poisson distribution cannot simply be read in an emission table, and thus has to be calculated instead. In line 8 of Listing 7 it can be seen how first the offset of the feature vector where the Poisson is found by stepping back from the end of the feature vector by the number of Poisson variables, and each probability is then calculated using an overload of the method that calculates a probability when given a lambda value and an observation value. The overloaded method is simply an implementation of the Poisson distribution calculation found in Equation 8. This achieves a compact and easily readable notation for calculating the probability of a state given the Poisson part of an observation sequence.

5.4 UPDATE RULES

The update rules for the elements of the HMM are pretty direct interpretations of the equations given in Chapter 3. As examples, the calculations for updating the transition matrix, and the λ values for the Poisson variables will be shown in this section.

In line 1 of Listing 8 a 2D array is declared to hold the values for the nominator contribution of the given time step of the replay. The nominator is then calculated in line 8, just as described in Section 3.1.2.2, with the only difference being the two-step calculation of the emission probabilities, due to the mixed nature of the observation values. The nominator is then normalised in line 18. The contribution is then added to the list of contributions across replays and time steps. In line 27, the belief state for the time step of the replay is calculated with *smoothing*. In line 30 this state belief vector is then

```
1  double[,] tmp = new double[States, States];
2
3  double scalingFactor = 0;
4  for (int s_t = 0; s_t < States; s_t++)
5  {
6          for (int s_previous_t = 0; s_previous_t < States; s_previous_t
               ++)
7          {
8                  tmp[s_t, s_previous_t] = Transitions[s_t, s_previous_t]
                       * GetEmissionProbability(s_t, replays[replayID][
                       timeStep]) * calculateProbabilityForPoisson(s_t,
                       replays[replayID][timeStep]) * fwd[replayID][
                       timeStep - 1, s_previous_t] * bwd[replayID][timeStep
                       , s_t];
9                  scalingFactor += tmp[s_t, s_previous_t];
10         }
11 }
12
13 for (int s_t = 0; s_t < States; s_t++)
14 {
15         for (int s_previous_t = 0; s_previous_t < States; s_previous_t
               ++)
16         {
17                 tmp[s_t, s_previous_t] = tmp[s_t, s_previous_t] /
                       scalingFactor;
18         }
19 }
20 //add matrix to collection of nominators across replays
21 transitionProbNom.Add(tmp);
22
23 //add forwardBackward for relevant timestep to denom sum.
24 double[] forwardBackwardPrevious = new double[States];
25 for (int s_t = 0; s_t < States; s_t++)
26 {
27         forwardBackwardPrevious[s_t] = fwd[replayID][timeStep - 1, s_t]
               * bwd[replayID][timeStep - 1, s_t];
28 }
29
30 forwardBackwardPrevious = NormalizeVector(forwardBackwardPrevious);
31
32 transitionProbDenom.Add(forwardBackwardPrevious);
```

Listing 8: The contribution to the update of the transition matrix for each timestep > 1 for each replay

```
1  for (int timeslize = 0; timeslize < T; timeslize++)
2  {
3          int observationOffset = Emissions.Count;
4          for (int poissonEmission = observationOffset; poissonEmission <
               Emissions.Count + poissonVariables; poissonEmission++)
5          {
6                  double[] forwardBackward = new double[States];
7                  for (int s_t = 0; s_t < States; s_t++)
8                  {
9                          forwardBackward[s_t] = fwd[replayID][timeslize,
                               s_t] * bwd[replayID][timeslize, s_t];
10                 }
11                 forwardBackward = NormalizeVector(forwardBackward);
12                 double[] nom = new double[States];
13                 double[] denom = new double[States];
14
15                 for (int StateNo = 0; StateNo < States; StateNo++)
16                 {
17                         nom[StateNo] = forwardBackward[StateNo] *
                               replays[replayID][timeslize][poissonEmission
                               ];
18                         denom[StateNo] = forwardBackward[StateNo];
19                 }
20                 lambdaNom[poissonEmission - observationOffset].Add(nom);
21                 lambdaDenom[poissonEmission - observationOffset].Add(
                       denom);
22         }
23 }
```

Listing 9: The contribution to the update of λ values for each replay

normalised to account for the scaling factor of the backward message, as described in Section 3.1.1. Once this has been calculated for all time steps in all replays, the values of all the contributions for the nominator are summed, and likewise for the denominator. The nominator is then divided component wise for the respective columns.

Listing 9 shows the update contribution of a single time step of a replay, for the λ parameters of the Poisson observations. In line 3 the offset in tte observation vector is set by referencing the number of multinomial variables. for each poisson emission the smoothed state probabilies are then calculated in lines 6 to 11. As per Equation 9 the smoothed state probabilities are used directly as the denominator for the time step in line 21, whereas the nominator is the smoothed state probability multiplied by the value observed for the observation in line 17.

Just as for the transition update, the nominator contributions are then summed across all replays and timesteps in the end, as is the denominator values. Finally, a componentwise division for each state yields the updated λ values for the observation.

Implementation issues

Using BWAPI is not problem free, due to the code injection nature of the API, as well as SC being a 16 years old game. This leads to a lot of tinkering being necessary in order to make BWAPI run on a given machine. This is unfortunate, since the documentation for the newest release of BWAPI, 4.0.1 beta at the time of this report, is rather sparse.

While the BWAPI documentation claims to have features for automatic starting of SC replays[2], this feature is either broken in version 4.0.1 beta, or not working as specified in the documentation. The end result is that automation of replays has to be handled through the SC menu system, using simulated keyboard input to activate the hotkeys for the menu, which is not considered a good idea [5]. Simulated keyboard input turned out to not be particularly smooth either, since the minimum time between keypresses had to be accomodated to the tolerance of SC, which is not documented anywhere, since SC is a proprietal system never intended for the things BWAPI does. This results in the delay between keypresses being a matter of trial and error to decide how short it can be cut, and varies from machine to machine as well, based on the hardware of the machine running the SC client. On top of this, the BWAPI client was unable to properly clean up the references for the keypresses across uses, leading to the game and BWAPI client crashing on the menu screen after each replay, which in turn lead to the implementation of a watchdog service being neccessary to restart both SC and the BWAPI client after each iteration. This discussion should serve as a warning to people intending to use BWAPI: Just like other hacking projects, new and exciting functionality can be added, but the nature of hacking can also cause unforseen problems.

6 EXPERIMENTS

To investigate the validity of the ideas presented in this project, two types of experiments are performed: One goal is to investigate whether a mixed variable PHMM can provide accurate predictions of the number of currently living units, and the other to make an analysis of the most likely path through the state space, and see if this path actually makes sense in the context of SC strategies. In order to do this, first the number of hidden states must be determined. When determining the number of states, two considerations must be kept in mind: The model's complexity and the likelihood of the learned models. The main variable to adjust is the number of hidden states in the model, since the more hidden states are in the model, the better the model can take special cases into consideration, thus increasing the model's likelihood. However, too many states not only leads to a model that is harder for a human to make sense of, and thus validate, it also increases the risk of *over fitting*. If too many states are in the model, the model can assign each state to a too specific scenario, which leads to the state almost never being achieved, which means the state only responds to very specific circumstances. If the model ends up with too many specialised states, it may end up being unable to recognise general patterns and only specialised ones. For this purpose the Bayesian information criterion (BIC) score can be used to evaluate the suitable number of states.

6.0.1 *Bayesian information criterion*

The BIC score is an alternative to performing *cross validation*. The primary arguments for using BIC instead of cross validation is the relative ease of calculating a BIC score, compared to cross validation, as well as the fact that cross validation requires models to be trained only on subsets of the training data. Since only 1400 replays in total were available for this project, and a part of these should be reserved for testing, it would be a relatively significant amount of training data that would be exempt from each model's learning, if cross validation was performed. Unfortunately the BIC score is an indirect evaluation, as opposed to the actual testing of cross validation, so reservations regarding the quality of information from the BIC may be appropriate.

For calculating the BIC score of a Poisson and multinomial mixed HMM, the following equation is used:

$$\mathrm{BIC}(Y|\lambda) = \mathrm{Log}_2(P(Y|\lambda)) - \frac{I - 1 + S \cdot S - 1 + S \cdot P + S \cdot M - 1 \cdot V}{2} \log_2(N) \tag{10}$$

where N is the total number of time steps used to train the model, I is the number of initial states, S is the size of the transition matrix, P is the number of Poisson variables, M is the number of multinomial variables and V is the number of values the multinomial observations can assume.

For the initial BIC calculations, fifteen models were trained with number of states in each interval, all with the same number of Poisson variables and multinomial variables, as are intended for the final tests. The results can be seen in Figure 20

states	avg likelihood	BIC
10	-82838,10215	-89071,9313
15	-80477,41227	-90187,7083
20	-79064,04093	-92488,9181
25	-78723,05644	-96100,6289
30	-69448,42615	-91016,8081
35	-77342,6447	-103339,95

Figure 19: BIC calculations for 15 learned models in each interval step

Using this sample as a guidepost, models were trained with a state number between 10 and 15 and each number of states was used for training 15 models. These were then be used to make a final evaluation of the most suitable number of states.

states	avg likelihood	BIC
10	-82838,10215	-89071,9313
11	-81233,18516	-88143,2585
12	-81.915,2459	-89511,0881
13	-80594,26359	-88885,3992
14	-80699,55807	-89695,5116
15	-80477,41227	-90187,7083

Figure 20: BIC calculations for 15 learned models in each interval step

Based on the result shown in Figure 20, the final state number was decided to be 11.

6.1 PREDICTION

To properly explore the accuracy of the model, different scenarios are tested. The first experiment will consist of predicting the number of live zealot, dragoon and reaver units in the final time step, T19 to investigate how well the states of the learned model fit the test data. Two other experiments will be made to predict the state of the game 60 and 90 seconds in the future. Graphs will be produced to show the average error in production, destruction, as well as the number of live units. This will be compared to the standard deviation for the data set of each unit, in an attempt to determine how much any inaccuracies are due to the model and how much data variance is the

cause. For these experiments, the learned model with highest log-likelihood and 11 states, was used to predict results in 400 replays that were not used for the training of models.

6.1.1 *Standard deviation*

The standard deviation is a measure of the average variation of a set of data. For these experiments, each unit type has had the standard deviation calculated for construction, destruction, and live units. This is relevant, since this tells us something about how closely the replays match the average pattern, which in turn allows for assessment of the expected prediction error.

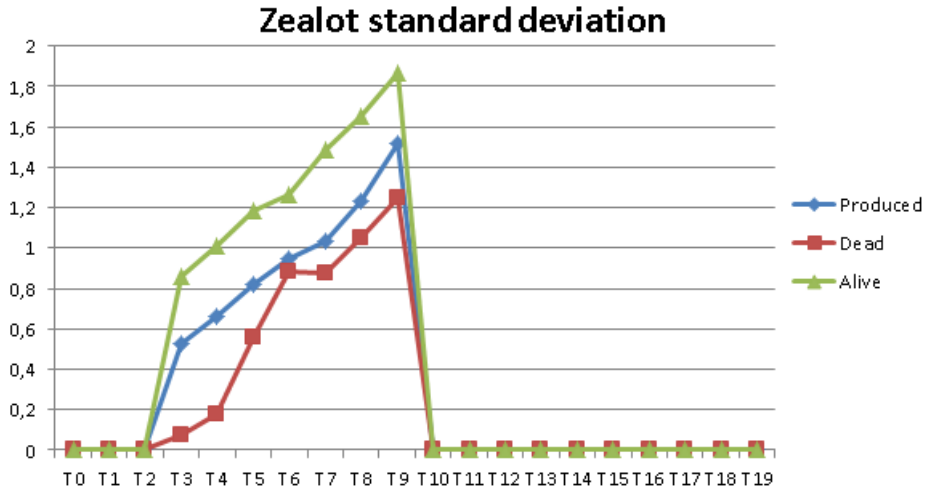

Figure 21: The standard deviation for the observed values, for the Zealot unit in each time step

As can be seen in Figure 21, the deviation for all zealot attributes becomes significant after time step T2. From this point on, we can expect the value of vary within the indicated interval, in the appropriate time step. Thus, we can expect the number of produced zealots in T3 to vary within 0.52. Similarly, Figure 22 and Figure 23 show the standard deviations for the dragoon and reaver unit. For Figure 22 it is worth noting that the *alive* and *produced* lines follow the exact same path.

6.1.2 *Average error with full evidence*

To get an overview of the accuracy of the model in accordance to the data, first experimental configuration is with full evidence until the final time step. Thus, the only prediction is the final time step, T19. A graph is produced to represent the average absolute error for each type of prediction.

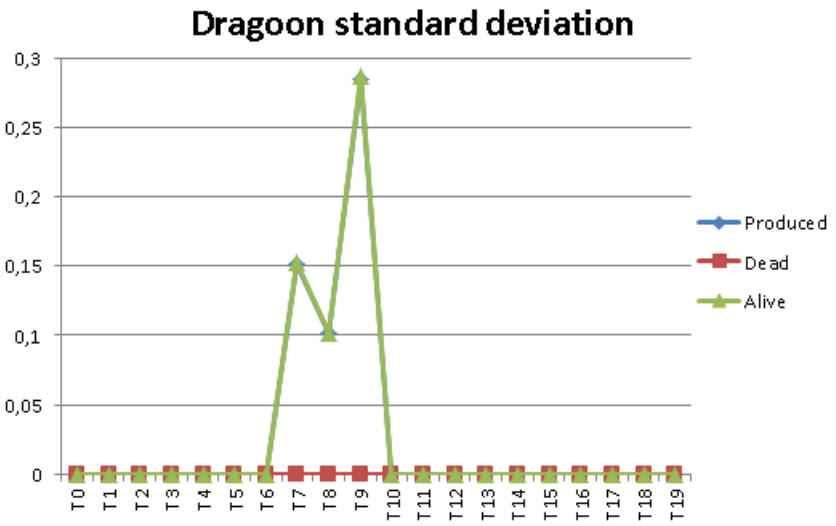

Figure 22: The standard deviation for the observed values, for the Dragoon unit in each time step

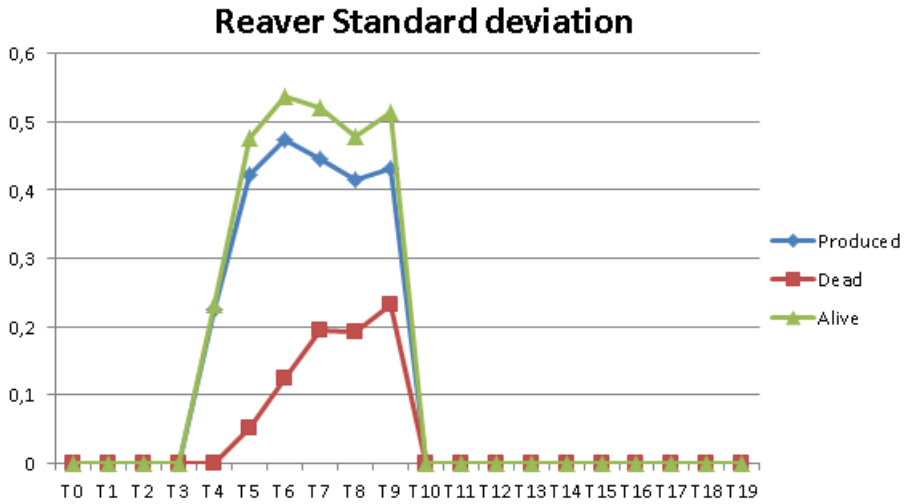

Figure 23: The standard deviation for the observed values, for the Reaver unit in each time step

6.1.2.1 *Production*

As can be seen in Figure 24, the average error for the production of zealots in this experiment configuration, the error for production, consistently lies below the standard deviation for production, as seen in Figure 21. This means that the accuracy of the prediction, on average, is better than the variance of the data by a factor of approximately a factor of 400 for zealots.

If we look at the similar value for the production of dragoons in Figure 25 we again see a consistently better result from the model, than the variance of the production value as is the case for the production of reavers in Figure 26.

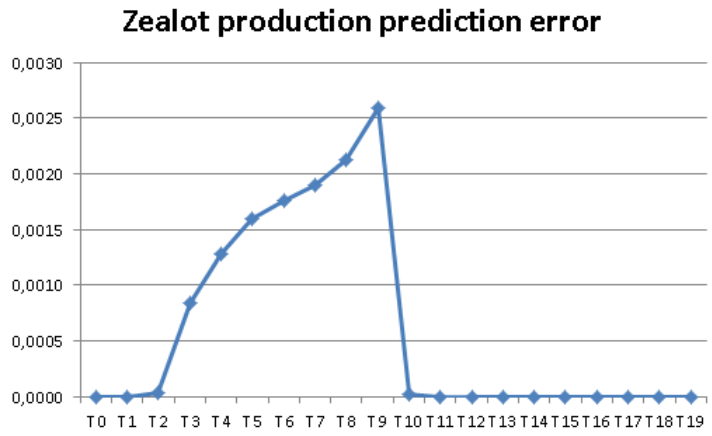

Figure 24: The average error for production of zealots, with full evidence

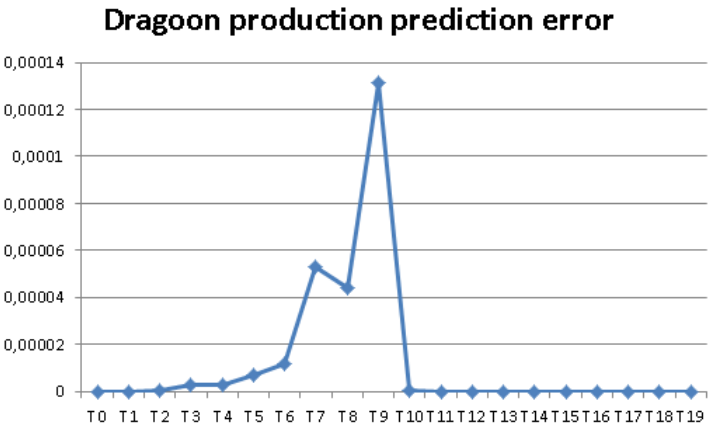

Figure 25: The average error for production of dragoons, with full evidence

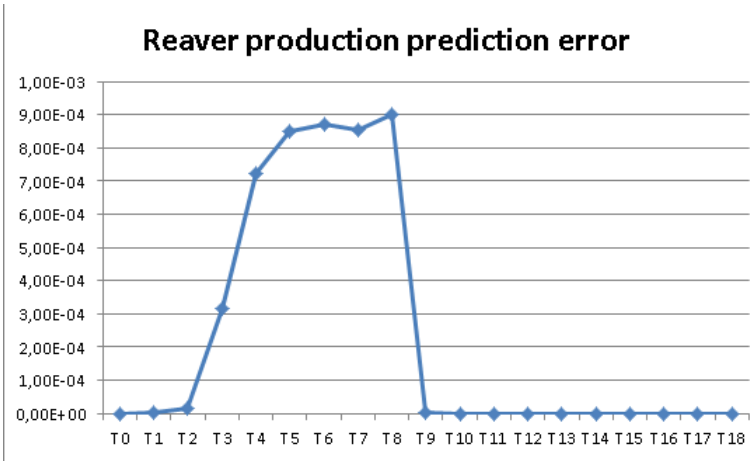

Figure 26: The average error for production of reavers, with full evidence

6.1.2.2 *Destruction*

In Figure 27 we can see that the average error for destruction of zealots is much higher that for production. The error is still smaller than the standard deviation by a large margin, but not as much as for production. The same thing is the case for destruction of dragoons in Figure 28 and reavers in Figure 29.

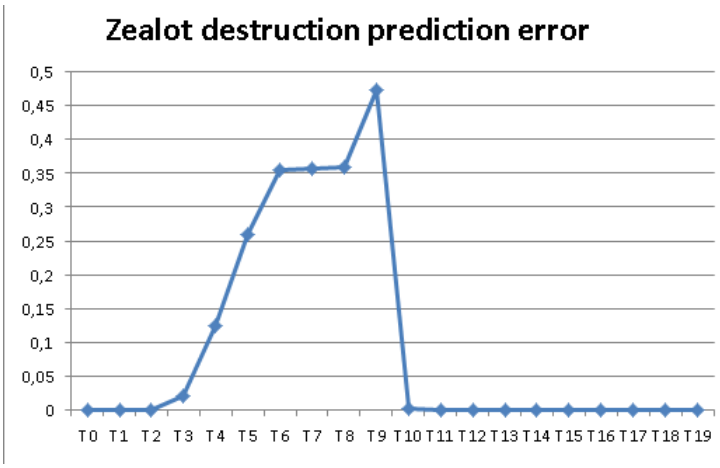

Figure 27: The average error for destruction of zealots, with full evidence

Figure 28: The average error for destruction of dragoons, with full evidence

The increase in error for all three types makes it a reasonable expectation that we will observe an even larger average error for the number of live units, since the prediction of the live number in a time step is calculated as $livenumberinthepreviousstimestep + (predictednumberofproduceduntis - expectednumberofdeadunits)$.

6.1.2.3 *Live units*

Looking at Figure 30, the assumption of an increase in error is indeed confirmed. The average error for the number of live zealots is only half the value

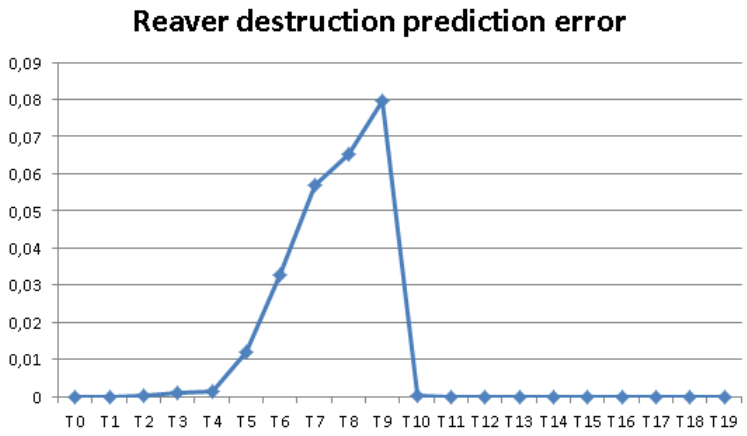

Figure 29: The average error for destruction of reavers, with full evidence

of the standard deviation. This still means that the variance learned by the model is descriptive of the patterns exhibited by the test set, but much less that is the case for production alone.

Figure 30: The average error for the number of live zealots, with full evidence

The number of live dragoons unfortunately exhibits a larger average error than the standard deviation, which means that we cannot fully attribute the error to data variance. The error is influenced by both the error margin for the production and destruction of dragoons, but it seems reasonable to attribute the error more to the destruction calculation, since the average error in each time step is approximately ten times higher for destruction than production.

The error for the number of live reavers reavers, shown in Figure 32 follows the same tendency as shown by the number of live zealots: an increase in average error, but still with less error than the standard deviation, which indicated that for these two unit types, the model matches the observed events better than the variance of the data on average, whereas the pattern for dragoons is less encouraging.

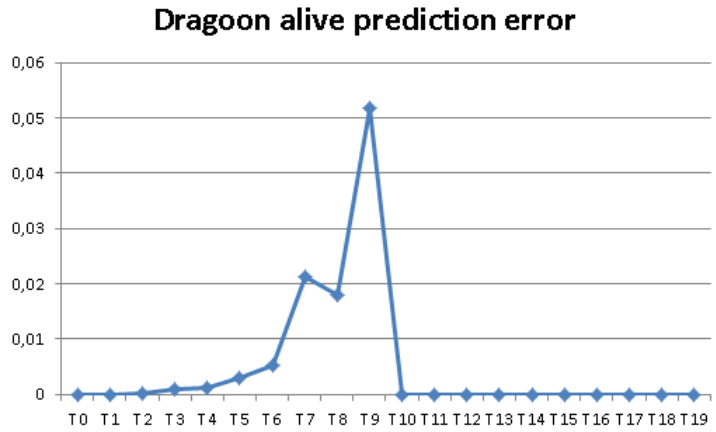

Figure 31: The average error for the number of live dragoons, with full evidence

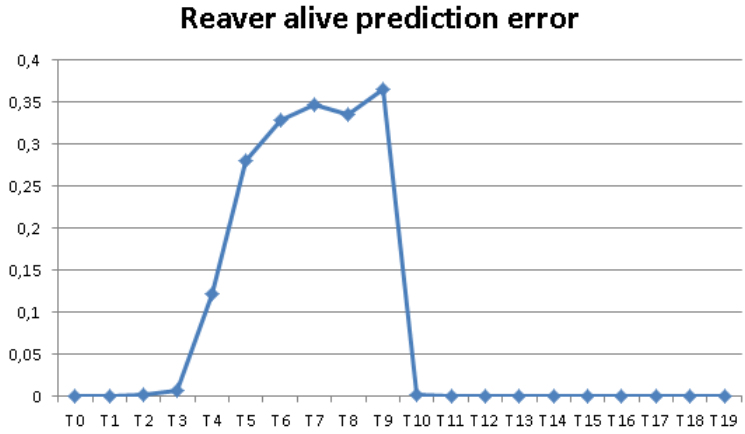

Figure 32: The average error for the number of live reavers, with full evidence

6.1.3 *60 second prediction*

In this configuration, prediction was applied to the 400 replays, where the model was tasked with predicting 60 seconds into the future, in each time step. Graphs of the average error were produced for both production, destruction and live numbers of each unit type.

6.1.3.1 *Production*

The average error for production of zealots, shown in Figure 33 is still significantly better than the standard deviation when predicting 60 seconds into the future, by a large factor. The error peaks in time step T5 which seems to insinuate that, once a certain criteria has been met, that the model recognises, the production of, or lack of production of zealots becomes very predictable.

Production of dragoons, shown in Figure 34 is very accurate. At the worst point in time, in T6, the average error in predicted production, is $1 * -10^6$, compared to the standard deviation for produced dragoons in T6 of 0.15.

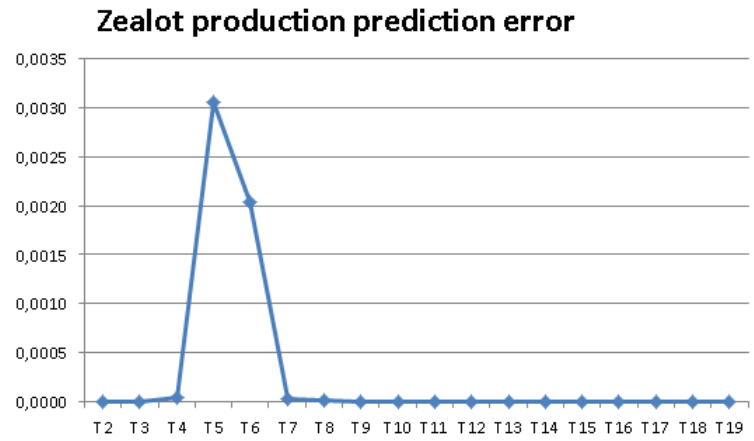

Figure 33: The average error for production of zealots, when predicting 60 seconds

This indicates that a production of dragoons can be predicted with a reasonable precision 60 seconds into the future.

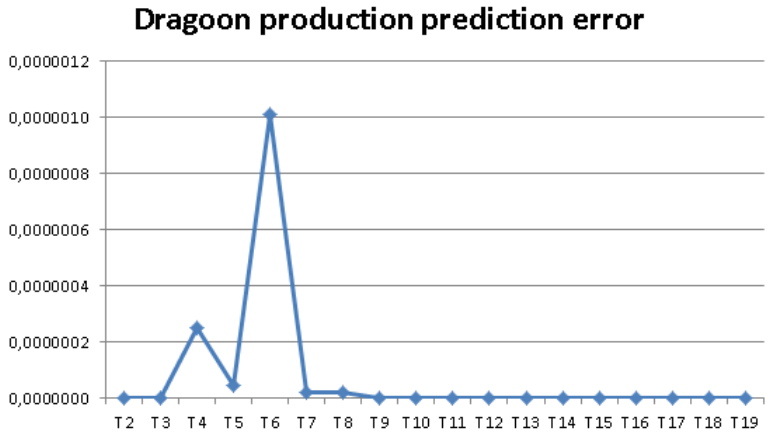

Figure 34: The average error for production of dragoons, when predicting 60 seconds

For reaver production, the largest error happens in time step T6, as can be seen in Figure 35. This seems to indicate that , as for dragoon production, once a certain criteria has been met, reaver production is very strictly patterned, especially when we look at the standard deviation for the production of reavers, which is approximately 0.45 in time step T7 and 0.4 in T8.

6.1.3.2 *Destruction*

Like for the full evidence error graph, it can be seen in Figure 36 that the average error for prediction of destruction is significantly higher, than for production. But just like the production of zealots, once a certain criteria has been met, the number of dead zealots in each time step can be predicted

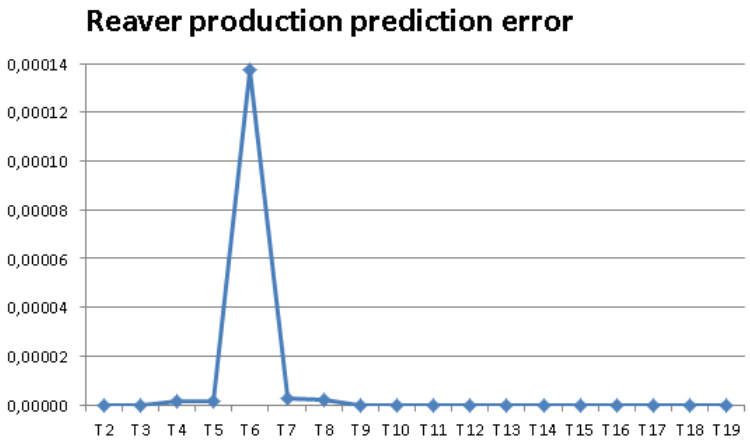

Figure 35: The average error for production of reavers, when predicting 60 seconds

with much greater accuracy. It is, however not the same event, since the error for production peaks one time step earlier than the destruction.

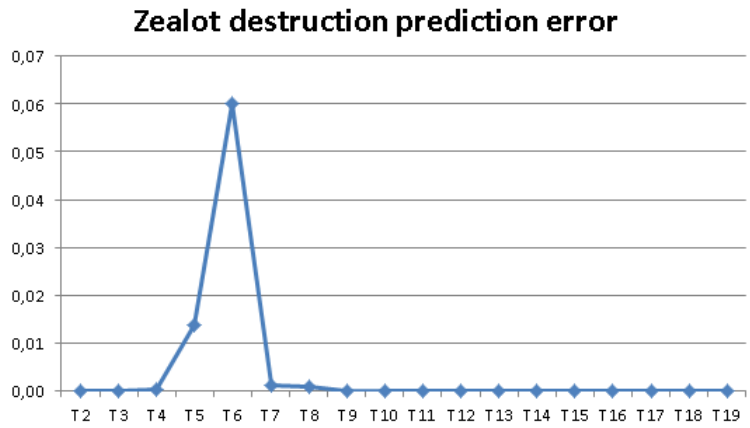

Figure 36: The average error for destruction of zealots, when predicting 60 seconds

When predicting destruction of dragoons, a better average error in each time step is actually achieved for 60 second prediction, shown in Figure 37, than for full evidence. This seems to indicate that the data set for the test is significantly different from the training set, in regards to destruction of dragoons.

For the reaver the average error in Figure 37 exhibits the same curious pattern of a smaller average error in each time step, when performing prediction across 60 seconds than when having full evidence in each time step.

6.1.3.3 *Live units*

For the prediction of live zealots, shown in Figure 39, the average error is generally better than the standard deviation, except for time step T5 where the error and deviation are very close in value. This seems to indicate that in

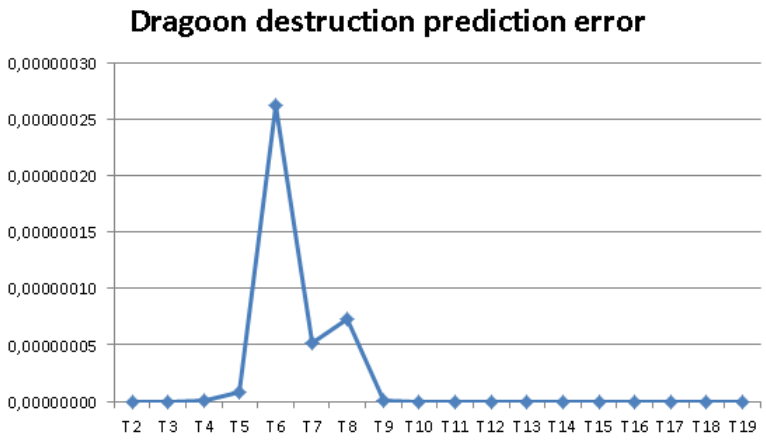

Figure 37: The average error for destruction of dragoons, when predicting 60 seconds

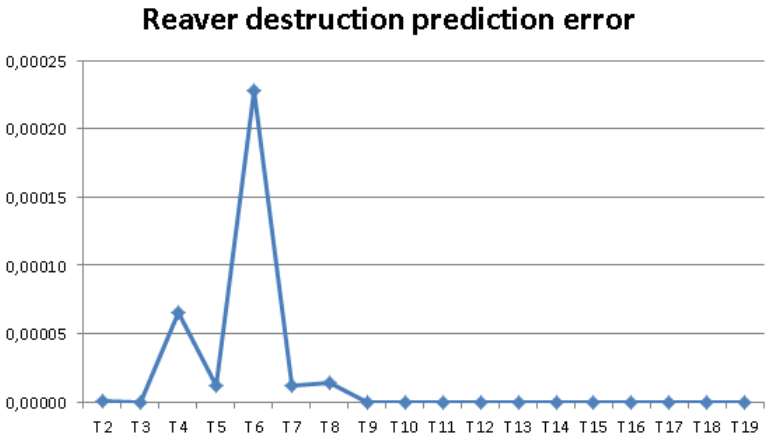

Figure 38: The average error for destruction of reavers, when predicting 60 seconds

time step T4, a distinction is made between states, that increases the accuracy of predictions.

When predicting for dragoons, the average error, shown in Figure 40 continues the unexpected behaviour of increased precision, compared to the full evidence across time steps. Once again the most likely explanation seems to be that there either must be a significant difference in the test data and the training data in the case of dragoons, or the model has learned something that does not correspond to reality.

For the number of live reavers, the average error is generally much lower lower than the standard deviation, as can be seen in Figure 41. This means that model predicts within a narrower margin than the natural variance of the data, in the case of reavers. A spike in average error occurs in T6, which is also where the standard deviation for the number of live units happens, so it seems reasonable to assume that these two properties are connected, since the actual value that is assumed is most uncertain in this time step.

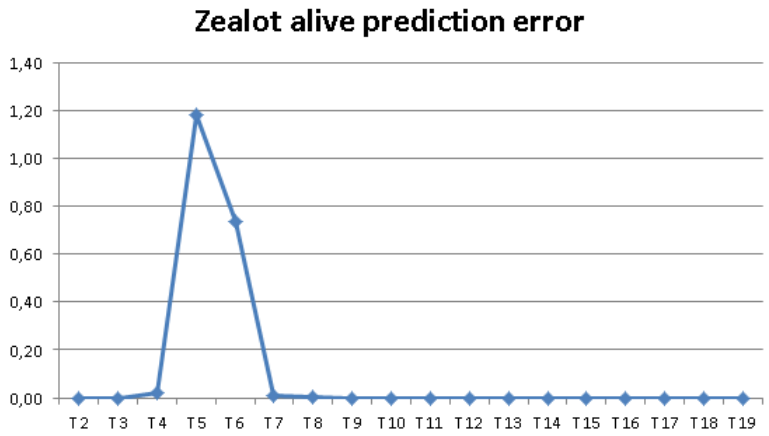

Figure 39: The average error for number of live zealots, when predicting 60 seconds

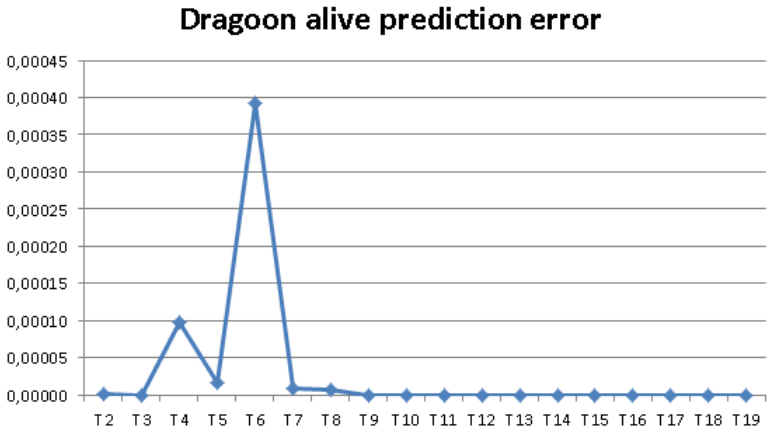

Figure 40: The average error for number of live dragoons, when predicting 60 seconds

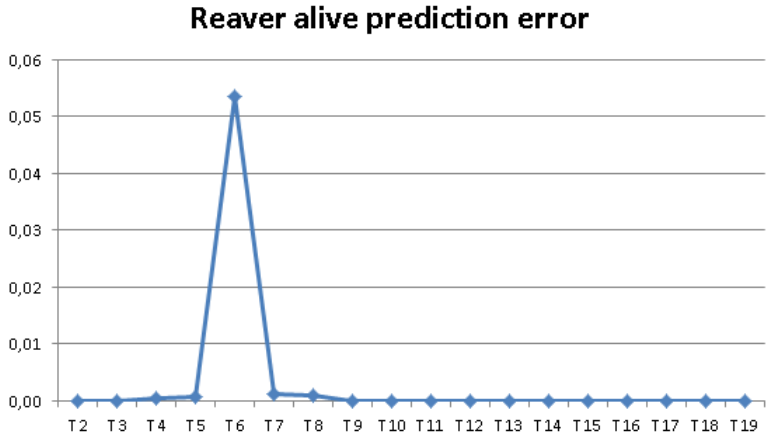

Figure 41: The average error for number of live reavers, when predicting 60 seconds

6.1.4 *90 second prediction*

In this configuration, prediction was applied to the 400 replays, where the model was tasked with predicting 60 seconds into the future, in each time step. Graphs of the average error were produced for both production, destruction and live numbers of each unit type.

6.1.4.1 *Production*

Zealot production generally follows the same pattern in Figure 42 as happened for prediction for 60 seconds, except with a larger average error, which is to be expected for the longer period covered without evidence.

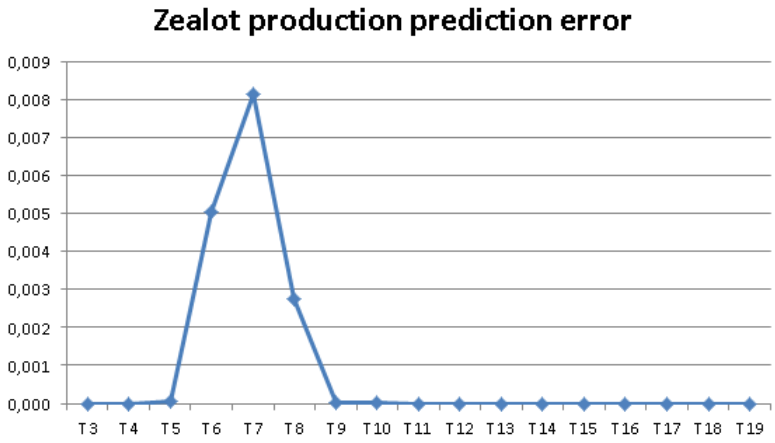

Figure 42: The average error for production of zealots, when predicting 90 seconds

The average error for prediction of dragoon production, in Figure 43, when predicting 90 seconds into the future, conforms more to the structure of Figure 34. The average error in each time step increases in comparison to the 60 second prediction, which is to be expected, since longer periods have to rely on only the learned transition probabilities.

The average error for prediction of reaver production in Figure 44 follows a similar pattern of error as the 60 second prediction, albeit with an average error that is approximately ten times higher in each time step, but still significantly lower than the standard deviation.

6.1.4.2 *Destruction*

Prediction of zealot destruction in a 90 second time frame, shown in Figure 45 shows an average error that is similar to the error exhibited by the 60 second prediction. However the error peaks at an earlier time step.

Similarly, to the zealot destruction, the average error for dragoon destruction, shown in Figure 46, follows the same overall pattern as the destruction for 60 prediction, with a higher average error in each time step.

The error for prediction of reaver destruction, shown in Figure 47 is a bit of an out-lier in this group with the highest average error spreading to

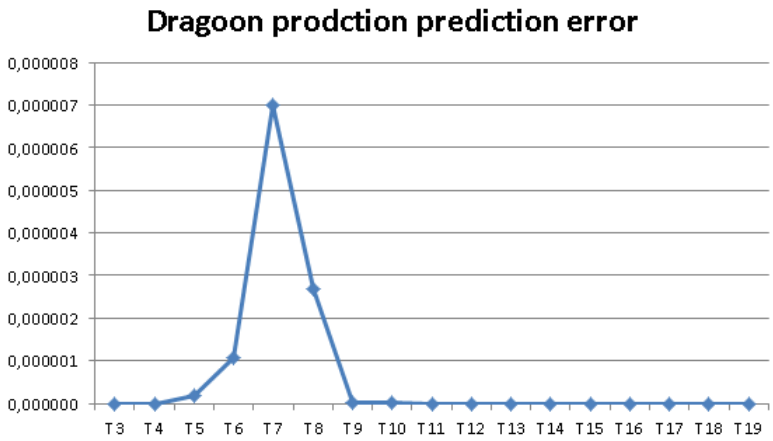

Figure 43: The average error for production of dragoons, when predicting 90 seconds

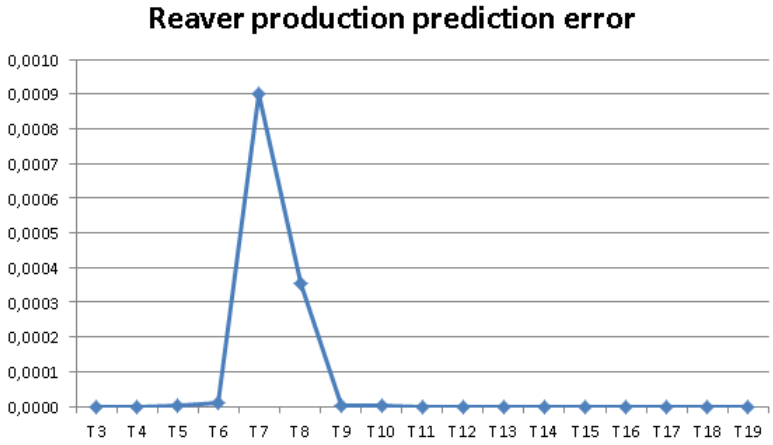

Figure 44: The average error for production of reavers, when predicting 90 seconds

two time steps, unlike the single time step of the other unit types. If we refer to the 60 second prediction however, this makes more sense as we can see that the early time steps offer some uncertainty about whether a reaver is destroyed in the following time steps, most likely until it becomes more certain if a reaver is built.

6.1.4.3 *Live units*

When predicting the number of live zealots, the average error in Figure 48, gets very close to the same as the standard deviation in time step T5. This means that it is not particularly useful trying to predict in this interval, since the replays naturally exhibit a variance equal to the error margin, which means that the model could predict that 1.2 zealots are produced in the time step, and no zealots could be produced at all.

The prediction of the number of live dragoons, shown in Figure 49 indicates that the model is generally good at predicting the number of live dra-

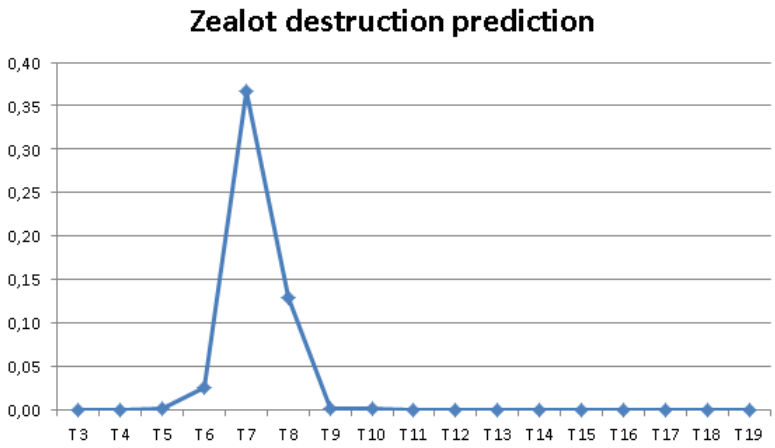

Figure 45: The average error for destruction of zealots, when predicting 90 seconds

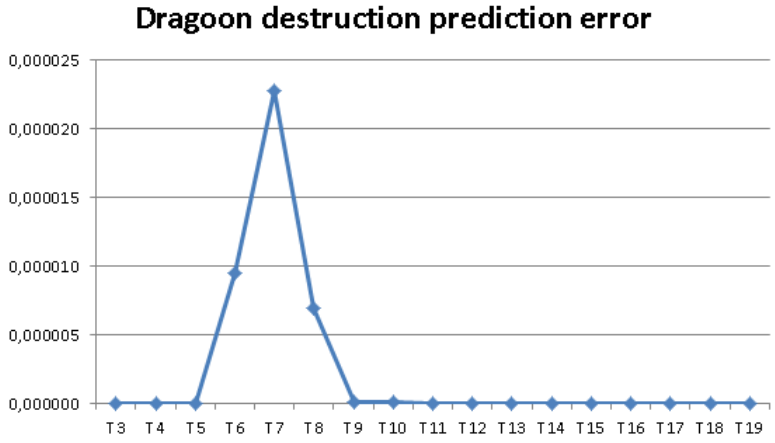

Figure 46: The average error for destruction of dragoons, when predicting 90 seconds

goons 90 seconds into the future. The standard deviation is approximately a thousand times higher than the prediction error, which means that the variance in predictions are much smaller than the natural variance of the observations.

The prediction of live reavers, shown in Figure 50, indicates that the model is somewhat capable of predicting the number of live reavers 90 seconds into the future. The average error is, at worst, 1/10 of the standard variation.

6.1.5 *Overall accuracy*

Overall the model seems to have a fairly good accuracy of prediction in regards to construction of units. Prediction of unit destruction, and thus prediction of the number of live units, could be better. The data analysis seems to indicate that, at least for dragoons, the problem could be that the number of events simply isn't very large, and thus don't confer much information

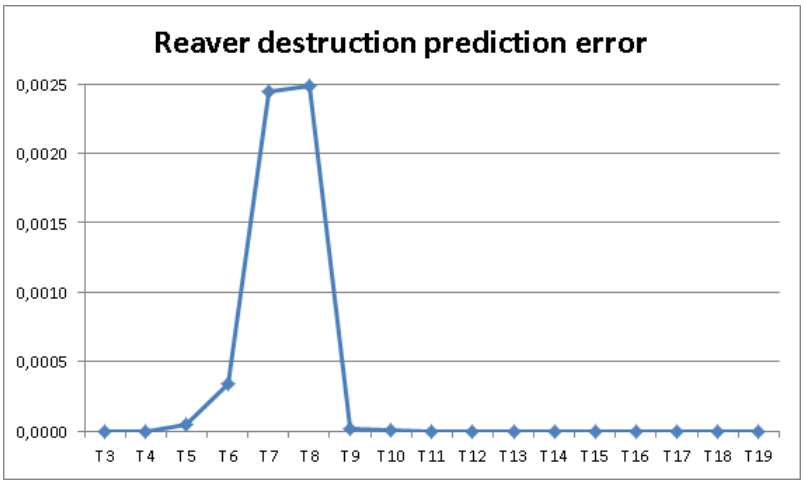

Figure 47: The average error for destruction of reavers, when predicting 90 seconds

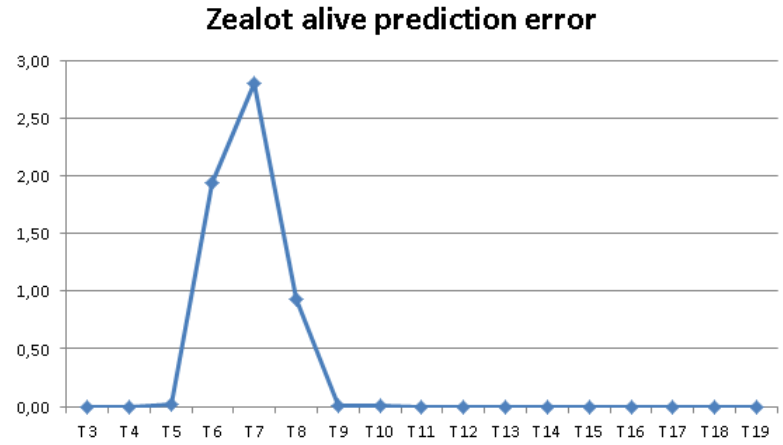

Figure 48: The average error for the number of live zealots, when predicting 90 seconds

about when this could take place. The generally good accuracy of unit production does however confer a good deal of precision to the accuracy of predicting live units. It is, however, odd that the average error for the full evidence sequences, particularly dragoons, have such high average errors, compared to the standard deviation. A possible cause for this could be the quality of the data, since no guarantees are given from the Team Liquid and GosuGamers sites about the quality of the replay. Adding to the murkiness of the data quality is the fact that both sites host replays from the ICC-Cup, which is a tournament for SC bots, which may vary extensively in playing ability. The model does however seem to be able to overcome this and learn to generally differentiate between this.

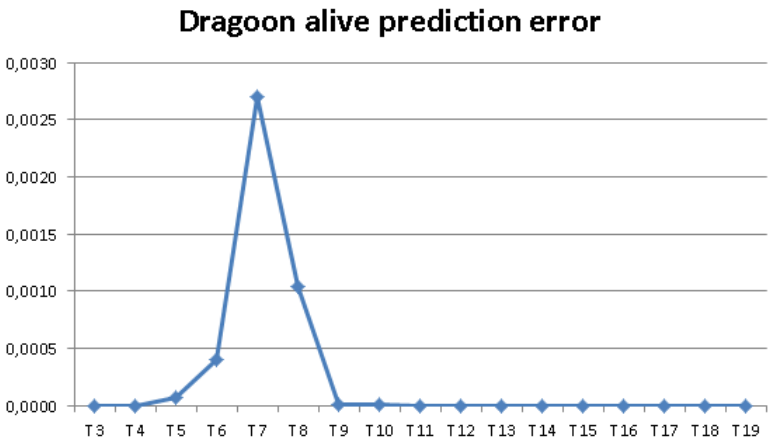

Figure 49: The average error for the number of live dragoons, when predicting 90 seconds

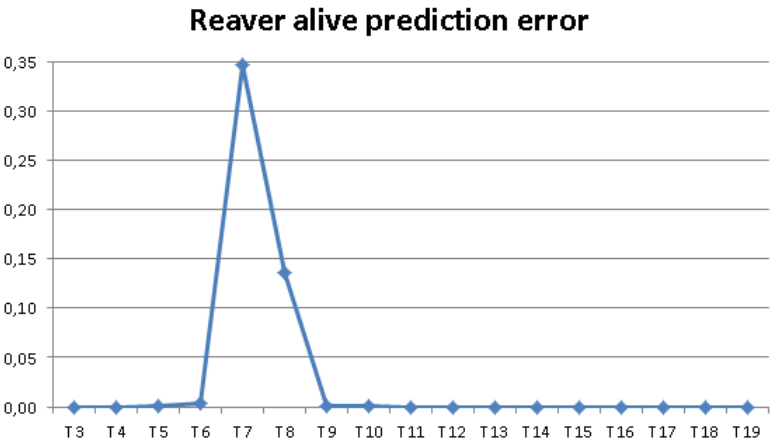

Figure 50: The average error for the number of live reavers, when predicting 90 seconds

6.2 STATE TRACE

The state chart has been produced using the 11 state model that had the highest log likelihood of −79025.43 with a transition threshold of 10%, a threshold for multinomial observations of 10%, and a threshold of 0.5 for λ values of Poisson variables.

Looking at the state chart in Figure 51 we can see that the learned model starts with a pattern much like what was expected, based on the discussion in Chapter 4: An initial production of approximately two probes, and production of a gateway in half the cases. It seems likely that the self-referring transition in State 2 will then be taken and a gateway will then be produced. This matches the pattern exhibited for production of gateway structures in the data analysis. From State 2 we can then move to State 7, which produces the cybernetics core with a high degree of probability. From State 7 we can

transition to State 1, which produces the approximately one dragoon that the data analysis indicated. Also in State 1 is the production of the Robotics facility, which enables the construction of the observatory structure, which happens 15.28% of the time in State 4. If the transition to State 4 is not taken, we can follow the self referencing transition in State 1, which allows for continued production of dragoons and also for the production of further infrastructure in the form of another gateway. If we transition from State 1 to State 4, dragoon production continues. In state 4 there is a 48% chance of remaining in State 4, and a 31.6% chance of transitioning to State 5 where the production of dragoons in increased. This follows the trends that Chapter 4 indicated. However, a pattern indicating that State 8 should exist was not seen, but the model seems to have learned it correctly if we look at the standard deviation for production and destruction in Section 6.1, this seems to be correct. This indicates that the model has learned what was expected in the data analysis chapter, as well as patterns that were not discussed in the data analysis, but makes sense when looking back at the results of the standard deviation.

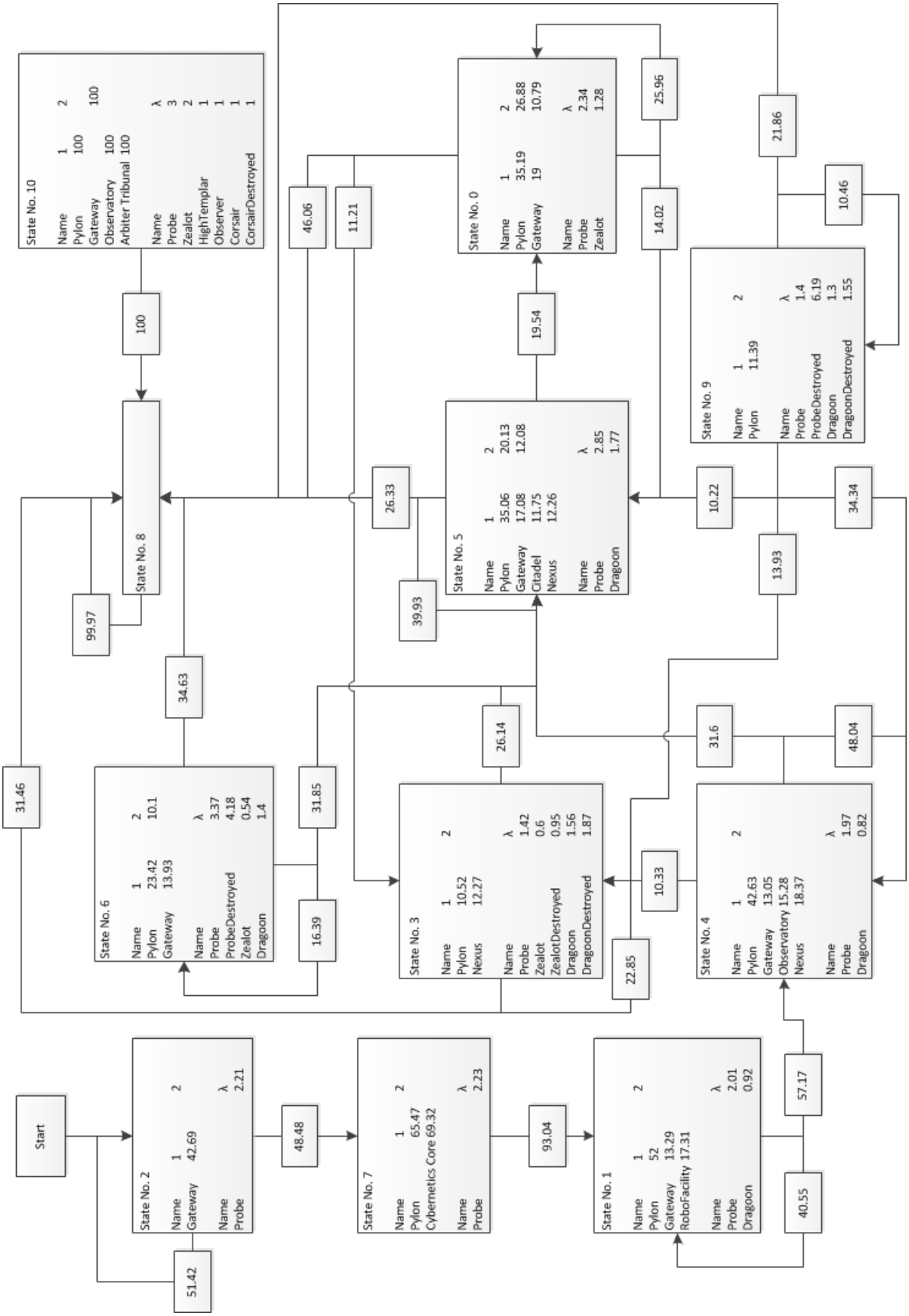

Figure 51: Statechart for model with 11 states

7 CONCLUSION

7.1 ACCURACY OF POISSON DISTRIBUTIONS FOR UNIT PATTERNS

Based on the combination of the state chart analysis, data analysis and prediction experiments, it seems reasonable to conclude that a Poisson distribution is a suitable representation of unit production in SC. The average error for prediction of production was very small compared to the standard deviation of the test data, and the state chart allows for a path that, with a reasonable probability, describes the expected production patterns for units. However, the concern presented in the data analysis remains regarding units that are of circumstantial use, such as the dark templar, since, as can be seen from the production of these being pruned from the state chart, they do not exhibit very high consistent production. The differentiation of describing buildings with multinomial variables does seem to make sense however, since many states in the state chart, for example, have a low, but consistent, probability of producing gateways. If this had been described as a Poisson variable, it seems very likely that these low probability productions of gateways would have been reduced to a very low average production value if represented as Poisson distributions.

The benefit of introduction of Poisson variables also seems to be dependent on the purpose of the model. If the purpose is to describe longer parts of a game, Poisson variables seem to make sense, but if the model only describes the opening stages of the game, multinomial variables seem to be better, since these are better for describing rare, or one-shot production types, such as was the case of zealot production in the seven minute interval.

7.2 LIMITATIONS OF MARKOV MODELS

Due to the probabilistic nature of Markov Models, errors start to creep in early in the predictions, even in the time steps where production of a unit has never begun yet. Though this error is small, the mere presence adds noise to the predictions. Another limitation is that, in order to learn a general model of the game state, a large pool of data needs to be available to train the model. Not only is this data pool not entirely trivial to accumulate in the case of SC. Another limitation is the training time. Learning a model that covers 10 time steps, and has 11 states took approximately 12 hours per model. Thus, training 15 models, in order to attempt a reasonable exploration of the optimisation surface, takes 180 hours of CPU time. While this can of course be parallelised, it is a limitation that is worth considering.

7.3 DATA QUALITY

Also worth noting is the importance of a decent quality level of the data used to train the model. State 8 of the learned model makes no sense from a domain knowledge perspective. There is absolutely no reason for a state where no production or destruction happens, and even less when the state has a 99.8% chance of transitioning back to itself. None the less, the data has not only taught the model that this was a necessity, but the experimental data has confirmed the sense in learning this state, at least seen from a data perspective.

7.4 FUTURE WORK

Some points remain open for future investigation at the end of this project:

- Investigation of the effect of modelling buildings as Poisson variables.
- Investigating the effect of having a single Poisson variable to describe the number of living units, instead of two separate variables that describe production and deaths.
- Investigating the impact of having some units that are expected to be produced for the duration of the game modelled as Poisson distributions, and having circumstantially produced units modelled as multinomial variables.

These points could all help the model become more accurate in describing certain edge cases, as well as possibly increase the overall accuracy, since for example a single variable to describe the number of living units of a type, would only have one variable that could contribute to the prediction error instead of the two that were present in the models of this project.

BIBLIOGRAPHY

[1] Accord.net framework. URL http://accord-framework.net/. Retrieved March 2, 2014.

[2] An API for interacting with StarCraft: Broodwar (1.16.1). URL http://code.google.com/p/bwapi/. Retrieved March 2, 2014.

[3] GosuGamers. URL http://gosugamers.net/general/. Retrieved November 2, 2012.

[4] Lordmartin Replay Ripper documentation. URL http://lmrb.net/doc.php?f=lmrr.htm. Retrieved December 16, 2012.

[5] Stackoverflow: How can i fire a key press or mouse click event without touching any input device. URL http://stackoverflow.com/questions/4057810/. Retrieved March 2, 2014.

[6] Team Liquid. URL http://www.teamliquid.net/. Retrieved November 7, 2012.

[7] Morten A. Nielsen, Stine B. Larsen, and Anders Hesselager. *Predicting Unit Production in StarCraft using Hidden Markov Models*. 2012.

[8] David W. Aha, Matthew Molineaux, and Marc Ponsen. Learning to win: Case-based plan selection in a real-time strategy game. In *in Proceedings of the Sixth International Conference on Case-Based Reasoning*, pages 5–20. Springer, 2005.

[9] David W. Albrecht, Ingrid Zuckerman, Ann E. Nicholson, and Ariel Bud. Towards a Bayesian Model for Keyhole Plan Recognition in Large Domains. In *In Proceedings of the Sixth International Conference on User Modeling*, pages 365–376. Springer-Verlag, 1997.

[10] Eugene Charniak and Robert P. Goldman. A Bayesian Model of Plan Recognition. *Artificial Intelligence*, 64(1):53–79, 1993.

[11] Ethan Dereszynski, Jesse Hostetler, Alan Fern, Tom Dietterich, Thao-Trang Hoang, and Mark Udarbe. Learning Probabilistic Behavior Models in Real-Time Strategy Games. 6:20–25, 2011.

[12] Jesse Hostetler, Ethan Dereszynski, Tom Dietterich, and Alan Fern. Inferring Strategies from Limited Reconnaissance in Real-time Strategy Games. 10, 2012.

[13] Ji-Lung Hsieh and Chuen-Tsai Sun. Building a player strategy model by analyzing replays of real-time strategy games. In *Neural Networks, 2008. IJCNN 2008. (IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*, pages 3106 –3111, june 2008. doi: 10.1109/IJCNN.2008.4634237.

[14] Finn V. Jensen and Thomas D. Nielsen. *Bayesian Networks and Decision Graphs*. Springer Publishing Company, Incorporated, 2nd edition, 2007. ISBN 9780387682815.

[15] Henry A. Kautz. A formal theory of plan recognition and its implementation. In *Reasoning about Plans*, pages 69–125. Morgan Kaufmann, 1991.

[16] Santiago Ontañón, Kinshuk Mishra, Neha Sugandh, and Ashwin Ram. Case-Based Planning and Execution for Real-Time Strategy Games. In Rosina Weber and Michael Richter, editors, *Case-Based Reasoning Research and Development*, volume 4626 of *Lecture Notes in Computer Science*, pages 164–178. Springer Berlin / Heidelberg, 2007. ISBN 978-3-540-74138-1. URL http://dx.doi.org/10.1007/978-3-540-74141-1_12.

[17] Roberta Proli, Giovanna Redaelli, and Luigi Sezia. Poisson hidden markov models for time series of overdispersed insurance counts. In *Colloquium International Actuarial Association-Brussels*, pages 461–472, 2000.

[18] Lawrence R. Rabiner. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. In *Proceedings of the IEEE*, pages 257–286, 1989.

[19] Sindhu Raghavan and Raymond J. Mooney. Abductive plan recognition by extending bayesian logic programs. In Dimitrios Gunopulos, Thomas Hofmann, Donato Malerba, and Michalis Vazirgiannis, editors, *Machine Learning and Knowledge Discovery in Databases*, Lecture Notes in Computer Science, pages 629–644. Springer Berlin / Heidelberg, 2011. ISBN 978-3-642-23782-9.

[20] Stuart J. Russell, Peter Norvig, John F. Candy, Jitendra M. Malik, and Douglas D. Edwards. *Artificial intelligence: a modern approach*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996. ISBN 0-13-103805-2.

[21] Gabriel Synnaeve and Pierre Bessiére. Bayesian Model for Opening Prediction in RTS Games with Application to StarCraft. *IEEE Conference on Computational Intelligence and Games*, 8:281–288, 2011.

[22] Ben G. Weber and Michael Mateas. Case-Based Reasoning for Build Order in Real-Time Strategy Games. In *Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2009)*, 10/2009 2009.

[23] Ben G. Weber and Michael Mateas. A Data Mining Approach to Strategy Prediction. 8:140–147, 2009.