Journal of AI and Data Mining
Vol 7, No 1, 2019, 149-159
DOI: 10.22044/JADM.2018.6932.1815

Robust Opponent Modeling in Real-Time Strategy Games using Bayesian Networks

A. Torkaman and R. Safabakhsh*

Department of Computer Engineering & IT, Amirkabir University of Technology, Tehran, Iran.

Received 03 April 2018; Revised 12 May 2018; Accepted 03 July 2018
**Corresponding author: safa@aut.ac.ir(R. Safabakhsh).*

Abstract
Opponent modeling is a key challenge in the Real-Time Strategy (RTS) games since the environment in these games is adversarial and the player is not able to predict the future actions of his/her opponent. Moreover, the environment is *partially* observable due to the fog of war. In this paper, we propose an opponent model that is robust to the existing observation noise due to the fog of war. In order to cope with the existing uncertainty in these games, we design a Bayesian network whose parameters are learned from an unlabeled game-log dataset so it does not require a human expert's knowledge. We evaluate our model on StarCraft, which is considered as a unified test-bed in this domain. The model is compared with that proposed by Synnaeve and Bessiere. The experimental results on the recorded games of human players show that the proposed model is capable of predicting the opponent's future decisions more effectively. Using this model, it is possible to create an adaptive game intelligence algorithm applicable to RTS games, where the concept of build order (the order of building construction) exists.

Keywords: *Bayesian Network, Opponent Modeling, Real-Time Strategy Games, StarCraft.*

1. Introduction

In the recent years, the game industry has improved the graphics of the games in order to make them more entertaining for human players. However, they have often neglected the use of new and efficient artificial intelligence (AI) techniques to create computer-controlled agents in the game. Instead, they have used the older and much simpler approaches such as the rule-based algorithms and finite state machines [1]. Using such non-adaptive techniques results in predictable behaviors that a human player can easily distinguish and exploit. Thus the game would not be challenging anymore and the AI opponents are not amusing enough to play against. In commercial games, the AI players often cheat to make the game more challenging. For example, in StarCraft, the AI player can see the entire map but the human player can see only the area close to his/her unit [1, 2].

However, researchers have applied opponent modeling to create human-like AI players and more challenging games. Opponent modeling is one of the main challenges in the real-time strategy (RTS) games [2-4]. The opponent model is an abstract representation of an opponent. In RTS games, a large number of available actions and a partially observable environment make searching for the optimal solution computationally costly. Hence, the opponent models facilitate finding the optimal solution in these games by reducing the search space size through changing the likelihood of certain solutions [5]. In order to construct a practical model, the player has to have some information about his/her opponent gathered from either the current play or the previous ones.

RTS games are one of the most popular types of video games that constitute a good test-bed for AI algorithms. The RTS games are simulations of real wars in which all players can perform actions simultaneously. Each player controls a team of workers, buildings, and military units in a specific map, and s/he should gather different resources to build units and buildings. Some units are workers whose task is resource gathering, while the others

are the military ones that defend or attack. A player cannot see the entire map. Each unit has a limited sight range in which s/he can observe his/her opponent units and buildings. Hence, the players have incomplete information about their opponents. This is called the *fog of war*. In addition, the size of the state space and the number of available actions in each state are very large. Particularly, the attributes such as uncertainty (due to the fog of war and unknown intention of the opponent) and very large state space present challenges to the opponent modeling problem [2, 6].

Due to this high complexity, the standard planning and decision-making algorithms used for solving classic board game problems, like game tree search, are not directly applicable to RTS games without defining some levels of abstraction [2]. Common abstraction levels are the 'strategy', 'tactics', and 'reactive control'.

In this paper, we present an algorithm that models the strategy of the opponent in RTS games, and particularly in StarCraft: Brood War. We present an opponent modeling approach for RTS games based on the Bayesian networks, which addresses the uncertainty existing in these games. The parameters of the proposed Bayesian network are learned from the recorded game logs.

The order of building construction is a strong indicator of the player's strategy [7] and can help to determine the strategy of the player [2]. In addition, to develop a good opponent model, some information about the order of building construction of the opponent is required [8]. Hence, we consider a variation of this order as the opponent model. In other words, we describe an opponent by his/her building construction order. The proposed method is an unsupervised technique that can predict the opponent buildings before they are constructed. The building construction order of an opponent can be used to predict his/her strategy [7, 9, 10]. The rest of this paper is organized as what follows. Section 2 reviews the existing studies in opponent modeling. We introduce the StarCraft game as our test-bed and propose our model in Section 3. The experiments performed on recorded games of human players to evaluate the proposed model are described in Section 4. Finally, the conclusions and future work are presented in Section 5.

2. Opponent modeling

This section surveys various opponent modeling approaches proposed so far. There exist six groups of opponent modeling techniques, as follow [11].

1. Evaluation functions

In heuristic search, the opponent model focuses on the player's preferences. The preferences are determined using an evaluation function. An evaluation function assigns a score, which is determined based on the player preferences to each state of the game. An opponent model can be made up of this evaluation function. This method has been used in computer chess [12]. An expert human player builds a dataset of the possible chess board states and their preferable actions. In order to maximize the evaluation function, its weights are updated by a linear discriminant method [12].

2. Classification and clustering methods

In classification methods, an expert human player determines the class of the opponents. Then each opponent's play style is labeled based on its attributes.

One example of this approach is given in the Webber and Mateas's method [13]. They use a large dataset of StarCraft game logs (game replays) to train different classifiers like the nearest neighbor and C4.5 decision tree to predict the strategy of the opponent. Other examples of the classification approaches are neural networks (Soccer game) [14], hierarchical classifier (Spring game) [15], instance-based learning (Poker game) [16].

The clustering methods recognize a set of possible categories of opponents using the recorded game information without the human's expert knowledge, for example, a quality threshold clustering method [17] and the K-means clustering method [18].

3. Rule-based models

These models are based upon some production rules, mapping conditions into actions. Dynamic scripting is one of these methods, which is used in commercial computer games [19].

4. Finite state machines

The finite state machine model is a collection of the states of the game. When a condition is satisfied, the state of the machine will change and a transition will occur. These models are a variation of the rule-based models. Machado et al. have used the finite state machine in First Person Shooter games [20].

5. Probabilistic models

Probabilistic models can deal with uncertainty; therefore, they are efficient tools for the games with incomplete information.

A Bayesian model is proposed to predict both the opponent build tree (state of the buildings

constructed by the player; we will define build tree in detail in Section 3.2) and the opponent opening strategy in StarCraft [7, 9, 21]. Another example of the probabilistic approaches is the hidden Markov model [22].

6. Case-based models

In these methods, a database of the game states and their corresponding actions is made using game logs. This database is called a case base. During the game, the most similar case to the current game state is chosen and its related action is performed by the player. Farouk et al. [23] have proposed a generic opponent modeling based on case-based reasoning (CBR) for the Spring game.

3. Proposed model

As mentioned in Section 1, we selected StarCraft:Brood War as our test-bed. Hence, before addressing the proposed opponent model, we introduce this game and discuss why we choose it as a test-bed.

3.1. StarCraft

StarCraft is a canonical RTS game released by Blizzard Entertainment in March 1998. Its first extension, StarCraft:Brood War, was released in November 1998. There exist three fictitious races, namely Terran (T), Protoss (P), and Zerg (Z), each of which has different kinds of units and buildings. In this work, we consider only one versus one (duel) mode of this game. As a result, there exist six different match-ups (race combination) as follow: PvP (Protoss *vs.* Protoss), PvT (Protoss *vs.* Terran or Terran *vs.* Protoss), PvZ, ZvZ, ZvT, and TvT. The order of players is not considered; thus in this paper, Protoss *vs.* Terran and Terran *vs.* Protoss are the same.

Each player has a team of units and buildings. Some units are workers and some are military ones. The buildings have different features, and produce different kinds of units. For example, Nexus building (the base building for Protoss race) produces Probes, the worker unit and Gateway produces Zealot, Dragoon, High Templar, and Dark Templar, the military units.

At the beginning of a game, a player has one building as the resource gathering center and some worker units. In order to build more units, buildings and army workers gather two different kinds of resources, namely minerals and Vespene gas. The player needs minerals to construct everything but gas is only used for the advanced buildings and units. Each player can construct his/her units and buildings in a pre-defined sequence, which is called the *technology tree*. This sequence unlocks new units, buildings, and research upgrades. Research upgrades improve the functionality of the player units. At the beginning of the game, a player chooses his/her build order, in which s/he would construct his/her buildings. The build order defines the player strategy [2].

We choose StarCraft:Brood War as a test-bed for our algorithm for the following reasons:

- It is a standard test-bed that has been recently used by many researchers [8, 9, 13, 21, 24-26].
- A Free and open source C++ interface, Brood War Application Programming Interface (BWAPI)[1], was released in 2009, which allows injecting the AI codes into the StarCraft:Brood War. Using this interface, the programmers can read all the relevant information of the game states and create a new AI player.
- This game provides a mechanism to record the game logs that are referred to as the game replays. The game engine can recreate the gameplay using them deterministically. There exist some websites, for instance, TeamLiquid[2], GosuGamers[3], and ICCup[4], which store large repositories of replays from professional gamers. Replay files are stored in a proprietary binary format, which is not well-documented. Thus in order to parse them, some free pieces of software are developed, for instance, LordMartin Replay Browser[5], BWChart[6], and bwrepdump[7]. These extensive replay repositories and the developed free pieces of software provide a great deal of domain knowledge from which the AI techniques can learn the opponent model and strategy.

3.2. Opponent model using Bayesian networks

Several aspects of the opponent can be modeled in RTS games. In this work, we focused on the recognition of the opponent's build order. The player observes some constructed buildings of the opponent (due to the fog of war, all buildings are not observable). Using the proposed method, the player can recognize the build order of his/her opponent and then predict his/her future buildings. As mentioned in Section 1, the order in which the opponent constructs his/her buildings can be used to determine his/her strategy [7, 8]; therefore, we consider this order as the opponent model.

[1] https://github.com/bwapi/bwapi
[2] http://www.teamliquid.net/replay/index.php
[3] http://www.gosugamers.net/starcraft/replays-archive
[4] http://iccup.com/en/starcraft/replays.html
[5] http://lmrb.net/
[6] http://bwchart.teamliquid.net/us/bwlib.php
[7] https://github.com/SnippyHolloW/bwrepdump

Practically, we use a variation of technology tree, which is called the build tree. The build tree notation was first defined by Synnaeve and Bessiere [21]. The build tree is a strong indicator of the player's strategy [7]. It is the "buildings" part of the technology tree with duplications of some important buildings; hence, it does not contain any researches or upgrades. StarCraft has 16 built-in buildings for the Protoss race such as Nexus, Pylon, Gateway, Assimilator, Forge, Shield Battery, Cybernetics Core, and Photon Cannon[8]. These buildings have different features and produce different units. In order to define the build tree, we use all these 16 buildings. In addition, duplications of some important buildings, namely Nexus, Pylon, and Gateway are inserted into the build tree. These duplications are the second construction of important buildings. For instance, Nexus2 is the second Nexus, which means that the player aims to expand his/her territory. Additionally, this building increases the resource gathering power of the player. Pylon provides for a player extra Psi, the gameplay counter-limiting the number of Protoss units. Hence, extra Pylon shows more production capacity of the player, which is noteworthy. The second Gateway increases units producing the power of the player. Considering these three duplications of buildings, we use 19 buildings to show build trees for the Protoss race.

According to the build tree definition, {Pylon, Gateway} and {Pylon, Pylon2, Gateway} are two different build trees but have the same technology tree.

We use a Bayesian network to construct the model of an opponent. Bayesian networks provide a powerful tool for solving decision-making problems under uncertainty. Thus they are able to handle the RTS game uncertainties. All information about the map and the opponent is not available for the player due to the fog of war and the opponents' intention, thus s/he cannot infer the state directly from the observation. In other words, the computation of $P(State|Observation)$ is not straightforward but $P(Observation|State)$ can be learned from the game logs, where there is no fog of war. Using the Bayes rule, the player can infer

$$P(State|Observation) = \{P(Observation|State)\} / P(Observation) \quad (1)$$

[8] For a complete list, see: http://wiki.teamliquid.net/starcraft/Buildings

Bayesian networks are graphical models that indicate probabilistic relations among some variables. They have two main components: the structure and the conditional probability distributions. The structure, a directed acyclic graph, determines the variable conditional dependencies. Random variables are represented with the graph nodes, and their conditional dependencies are represented with the paths through the edges. The conditional probability distribution for each node in the graph specifies its probability conditioned on its parents' values. Figure 1 shows our proposed Bayesian network. In the following sub-sections, we define the variables and conditional probability distributions of the Bayesian network.

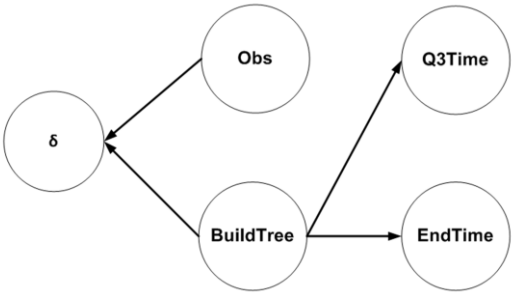

Figure 1. Bayesian network for Build tree prediction.

3.3. Variables

The variables are defined as what follow.

BuildTree: It is a set of buildings each player constructs in each game.

$$BuildTree \in \{\{b_1\},\{b_2\},\{b_1,b_2\},...\} \quad (2)$$

b_i s are different buildings of the given race. $AllBTs = \{\{b_1\},\{b_2\},\{b_1,b_2\},...\}$ shows all the possible build trees of the given race in each match-up.

For example, for the Protoss race, it is: {{Nexus}, {Nexus, Pylon},{Nexus, Pylon, Gateway},…}}. According to the technology tree and game rules, there exist 500 to 1600 possible values for the build tree depending on the race (without considering duplicate buildings) [21]. All of these build trees are not used in a competitive match. Hence, we consider only the build trees that are seen in the training dataset. We will discuss the training dataset in Section 4.1. In this work, we model the opponents of Protoss race playing against all the other races. In other words, we consider the PvP, PvZ, and PvT match-ups. For each match-up, we extract all the possible build trees (AllBTs) for Protoss race according to the training dataset. There exist about 330 different build trees for Protoss in PvP match-up, about 430 in PvT and about 420 in PvZ match-up.

We carry out a 10-fold cross-validation test. In each round of the cross-validation test, AllBTs is constructed according to the training partition of the dataset so it is independent from each test case, and does not change during each round of the test. Thus it is fixed for all the test cases. As a result, we can consider this value as a pre-defined value in the test phase. We will discuss the 10-fold cross-validation test in Section 4.2.

Obs: It is a binary sequence of building observations during gameplay. Thus it changes during gameplay. $Obs_{i=1..N}$ is 1 if the player has observed the i^{th} building of the opponent. According to the comment on the $BuildTree$ variable, the Obs variable shows the sequence of buildings that are observed during each test case.

$\delta \in \{0,1\}$ checks the compatibility of build tree value and observations.

EndTime and Q3Time: They are the times of gameplay. Time is expressed in game frames. Each second in StarCraft is divided into 24 frames. Then the current time of the game is the number of the game frame that is passed up to now. *EndTime* is the time when the build tree's last building is constructed, and *Q3Time* is the third quartile (Q3) of all buildings' construction time in a build tree. The justification for choosing the third quartile among others is given in Section 4.3.

The joint distribution of the above variables is specified as follow:

$$\begin{aligned} &P(BuildTree, Obs_{1:N}, \delta, EndTime, Q3Time) = \\ &P(BuildTree)P(Obs_{1:N})P(\delta \mid Obs_{1:N}, BuildTree) \\ &\times P(EndTime \mid BuildTree)P(Q3Time \mid BuildTree) \end{aligned} \quad (3)$$

3.4. Conditional probability distributions

The conditional probability distributions of this Bayesian network and their learning methods are defined as follow:

$P(BuildTree)$ shows the prior distribution of the build trees. $BuildTree$ is a discrete variable; thus to estimate its prior distribution, we calculate the occurrence frequency of each build tree in replay dataset. To learn *P(BuildTree)* (the prior distribution of $BuildTree$), we gather all the possible build trees occurring in the replay database per 3 match-ups (PvP, PvT, and PvZ). Since there is no fog of war in the training mode, we can learn *P(BuildTree)* by computing the frequency of its possible values.

$P(Obs_{1:N})$: For simplicity, we assume that it has a uniform distribution.

$P(\delta \mid Obs_{1:N}, BuildTree)$ is a Dirac delta function. It checks whether or not it is feasible to observe the set of specific buildings according to the given build tree. If the corresponding set of observed buildings in $obs_{1:N}$ is the subset of the build tree, $P(\delta = 1 \mid obs_{1:N}, buildTree)$ is equal to one; otherwise, it is equal to zero.

$P(EndTime \mid BuildTree)$ is the distribution of the time when the build tree is in a particular state. In other words, it is the distribution of the time when the build tree's last building is constructed. We assume that it is a Gaussian distribution whose parameters (μ and σ^2) will be learned from the replay dataset. In other words, a Gaussian distribution is fitted on the last building's construction time.

$P(Q3Time \mid BuildTree)$ is the Gaussian distribution of the third quartile (Q3) of all buildings' construction time in a build tree. The computation of its parameters is the same as that of $P(EndTime \mid BuildTree)$. We choose the third quartile among other quartiles to increase the accuracy of the proposed model. We perform two experiments to determine how many points in time and where are needed to achieve the best accuracy. Details of the related experiments are shown in Section 4.3.

Investigating the replay dataset revealed that the variables $P(Q3Time \mid BuildTree)$ and $P(EndTime \mid BuildTree)$ have a bimodal distribution for some (not all) values of the build tree variable. Thus determining the accurate distribution for this variable is not straightforward. We will try to estimate the distribution of these two variables by a precise distribution in our future work.

Our proposed model is somewhat similar to that of Synnaeve and Bessiere [21]. They have used the Bayesian inference to recognize the build tree of the opponent as well. This section shows the difference between our approach and the Synnaeve and Bessiere's approach [21] in how the parameters of the Bayesian network are learned. Their method suffers from two weaknesses, which we address as follow:

- They assume that the prior distribution of *BuildTree* is uniform. This is an unrealistic simplification. Additionally, some useful information would be lost by assuming uniform distribution. The more a build tree is repeated in the training dataset, the more probable it is to be seen in the test phase. Therefore, we calculate the prior distribution of the *BuildTree* variable

by counting the occurrence of its values in the replay data.

- In order to recognize the build tree given some observations, Synnaeve and Bessiere [21] check which build trees are the superset of the observations. Then they choose the one with the greatest PDF (Probability Density Function) value over the last building's construction time. In other words, they use only one point in time to recognize the actual build tree. Consequently, a change in construction time may lead to thewrong prediction of the build tree given the observations.

For further clarification, consider the following example: Let {Nexus (0), Pylon (1225), Gateway (1917), Gateway2 (2561), Pylon2 (3087), Nexus2 (6799)} be the player observation. The numbers in parenthesis show the construction time in frame resolution. The Synnaeve and Bessiere's approach recognizes the following build tree given this observation: {Core, Gateway, Gateway2, Nexus, Nexus2, Pylon, Pylon2}. This build tree is predicted incorrectly (the Core building is additional) due to the construction time of Nexus2. There exists a large delay between Nexus2 and its previously constructed building, which makes the algorithm to assume incorrectly another building was constructed in-between. Our model recognizes the following build tree, which is correct: {Gateway, Gateway2, Nexus, Nexus2, Pylon, Pylon2}. The prediction of our model is correct because our model uses two points in time to recognize the actual build tree, the construction time of Pylon2 and Nexus2 in this example. The construction time of Pylon2 acts as a guide point that enables the model to avoid mistakes. Thus the comparison of different build trees must be done at more than one point in time so as to make a better prediction. However, experiments show that when the fog of war (simulated by additional noise) exists, using more than two points in time would decrease the prediction accuracy (Section 4.3). Hence, we compare build trees in exactly two points in time including the last buildings' construction time and the third quartile of all buildings' construction time.

3.5. Inference

Once we have the joint distributions of the variables, we can answer any query about the value of a single variable by marginalizing over the others. In order to determine the model of the opponent, we should recognize his/her build tree given the observations. In order to find the value of *BuildTree* that maximizes the posterior probability

$$P(BuildTree \mid Obs_{1:N} = obs_{1:N}, \delta = 1, EndTime = t, Q3Time = t_3),$$

we ask a Maximum A Posteriori (MAP) query as follows:

$$\begin{aligned} &\arg\max_{BuildTree} P(BuildTree \mid Obs_{1:N} = obs_{1:N}, \delta = 1, EndTime = t, Q3Time = t_3) \\ &= \arg\max_{BuildTree} \{P(BuildTree) \times \\ &P(obs_{1:N}) P(\delta \mid obs_{1:N}, BuildTree) P(t \mid BuildTree) P(t_3 \mid BuildTree)\} \end{aligned} \quad (4)$$

When the opponent's build tree is recognized, his/her strategy can be predicted using another Bayesian network [7, 9, 10].

4. Experimental results

Due to the complexity of RTS games, designing a game intelligence capable of playing the entire game consists of very diverse sub-problems. This diversity makes the comparison of different studies very hard and scarce. To the best of our knowledge, there is no numerical comparison between various studies in the prediction of building construction order, and this paper is the first one that compares models to evaluate their accuracy in build tree prediction based on the quantitative measures.

In this section, we first describe the test dataset, and then introduce the evaluation metrics. The experimental results are next presented and compared with those of Synnaeve and Bessiere [21]. Finally, a discussion on how the time parameter should be set is given in the last sub-section.

4.1. Dataset

We use the replay dataset presented by Synnaeve and Bessiere [27] for the dataset. This dataset consists of 7649 recorded replays of professional StarCraft human players downloaded from TeamLiquid, GosuGamers, and ICCup. They parse replay files using bwrepdump9 that outputs three different file formats for each replay:

- Replay General Data (*.rgd) includes the players' names, map's name, and all game events including creation, discovery, attack, etc.
- Replay Location Data (*.rld) includes the location of the units.
- Replay Order Data (*.rod) includes all orders given to the units.

In this work, we use the .rgd files. The accuracy of the proposed method is evaluated on three different match-ups: Protoss *vs.* Protoss (PvP), Protoss *vs.* Terran (PvT), and Protoss *vs.* Zerg (PvZ). In other words, we model the opponents of

[9] https://github.com/SnippyHolloW/bwrepdump

Protoss race playing against all the other races. This race selection is done to simplify the implementation. We posit that if we can predict build tree for a specific race, we can do it for others as well. We use the .rgd files that contain only two player names. Hence, 411 game replays for PvP, 2098 for PvT, and 1788 for PvZ remain. The PvP match-up is a mirror one so each replay is used two times, one for each player perspective, which means that there exist 822 records for PvP.
It is worthwhile to say that due to some problems with the LordMartin Replay Browser, we use a different replay dataset than that used by Synnaeve and Bessiere [21]. They worked with a dataset of 8806 replays of highly skilled human players.

4.2. Results

In order to estimate the performance of the proposed model, we carry out a 10-fold cross-validation test. In each round of the cross-validation test, the replay dataset is partitioned into 10 complementary subsets. The opponent model is learned from 9/10 of the dataset, and is evaluated with the remaining 1/10.
We use the distance between the predicted build tree and the actual one to measure the accuracy of our model. This metric that is defined by Synnaeve and Bessiere [21] is computed as:

$$\begin{aligned} d(BT_{actual}, BT_{predicted}) &= card(BT_{actual} \Delta BT_{predicted}) \\ &= card\{(BT_{actual} \cup BT_{predicted}) \\ &\quad \setminus (BT_{actual} \cap BT_{predicted})\}, \end{aligned} \tag{5}$$

The $card(.)$ is the cardinality function of the set, and "\" is the set difference.
A distance of one means that there exists one less or one more building in the predicted build tree, and a distance of two means that one building is replaced in the predicted build tree or two less or two more buildings in the predicted build tree.
We evaluate our model using two different types of experiments. First, the accuracy of the model is measured at the current time of the game. In other words, up to $EndTime = t$, the player observes the constructed buildings of his/her opponent (BT_{actual}). Thus s/he can define $Obs_{i=1..N}$ according to his/her observations. The proposed model predicts the most probable built tree (among others in the $AllBTs$ set) according to its parameters ($BT_{predicted}$). The prediction power of the model is measured by $d(BT_{actual}, BT_{predicted})$. Let us define k as the number of opponent's buildings that our model can predict in advance. Hence, we call this experiment d for $k = 0$.
Secondly, the accuracy of the model is measured based on the prediction of future buildings. When the opponent's build tree at the current time (based on $Obs_{i=1..N}$ variable) is recognized, we can look ahead from now. We suppose that the opponent will construct his/her next buildings according to the recognized build tree. Hence, we can predict his/her next decision about the construction order. In order to evaluate the value of prediction about the future, we compute how many buildings (k) we can predict in advance at a fixed distance (d). In other words, to predict future buildings of the opponent, we ask MAP query about the $BuildTree$ variable at time $t+k$:

$$\begin{aligned} \arg\max_{BuildTree} P(BuildTree^{t+k} \mid Obs_{1:N} = partial(obs_{1:N}), \\ \delta = 1, EndTime = t + k, Q3Time = t_3 + k) \end{aligned} \tag{6}$$

It is worthwhile to notice that the $BuildTree$ variable is a set so it has no order. When we say that our model can predict k buildings in advance, we mean the set of k buildings, not their order. In order to predict exactly the next building, one can set $d = 1$ and $k = 1$. However, the purpose of this experiment is to measure how big k is to gather more information about what the opponent will be doing.
In order to simulate incomplete information due to the fog of war, we add random noise to the Obs variable from 0% to 80%. We randomly remove some building observations. It means that we set Obs_i to zero if the i^{th} building of the opponent is chosen to remove due to adding noise. Hence, 0% noise means that no building is removed and 80% noise means that 8 buildings out of 10 are removed.
In the first experiment, we compute the distance between the conjectured build tree and the actual tree up to the current game time. In other words, we compute d for $k = 0$ and run this test at three different times: 5th, 10th, and 15th minute of the game to consider all the game phases (early, mid, and late).
As mentioned in Section 3, the proposed approach is similar to that of Synnaeve and Bessiere [21]. Hence, we compare the performance of our model with theirs.
Figures 2 to 4 show the average values of d ($k=0$) for the Synnaeve and Bessiere [21] model and ours for the PvP, PvT, and PvZ match-ups, respectively.
Figures 2 to 4 show the noise impact on the distance to actual build tree in different game

phases. Our model results are shown in solid lines, and the result of the Synnaeve and Bessiere model are illustrated with dotted ones. The increases in the distance to actual build tree are much slower than the increase in the noise ratio for both models.

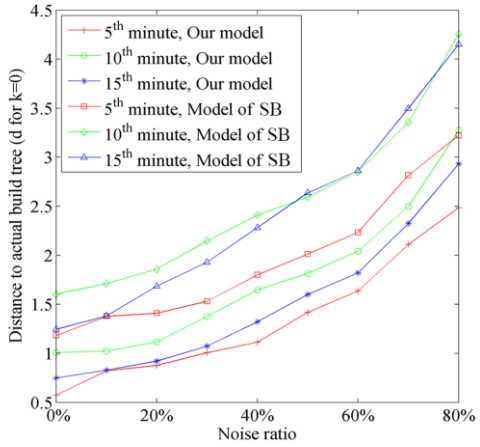

Figure 2. Distance to the actual build tree (d for $k=0$), PvP Match-ups (SB is Synnaeve-Bessiere [21]).

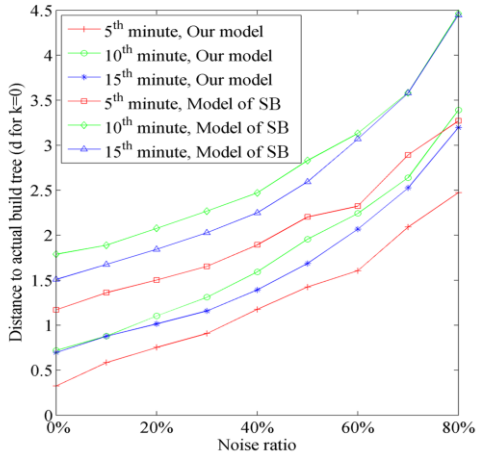

Figure 3. Distance to the actual build tree (d for $k=0$), PvT Match-ups (SB is Synnaeve-Bessiere [21]).

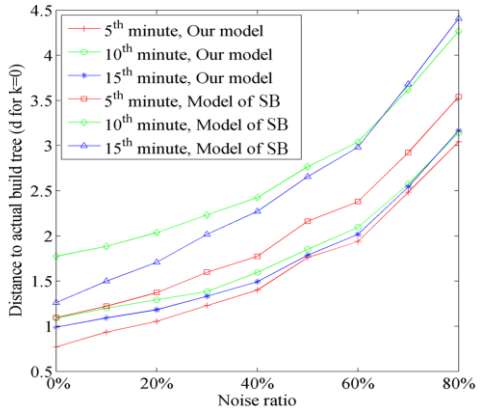

Figure 4. Distance to the actual build tree (d for $k=0$), PvZ Match-ups (SB is Synnaeve-Bessiere [21]).

As illustrated in figures 2 to 4, for all the three different match-ups, our model's average distance to actual build tree is lower than that of Synnaeve and Bessiere. The distance is reduced because we represent the build tree variable by a more realistic distribution. In addition, we adjust the time parameter to the exact one. As illustrated in figures 2 to 4, for all the three different match-ups, our model's average distance to actual build tree is lower than that of Synnaeve and Bessiere.

The distance is reduced because we represent the build tree variable by a more realistic distribution. In addition, we adjust the time parameter to the exact one.

In order to investigate the increase in the distance around the 10th minute, we plot the distance in the interval 5th to 15th minutes for the PvP match-up with 30% noise. As figure 5 shows, the trends are similar for both models.

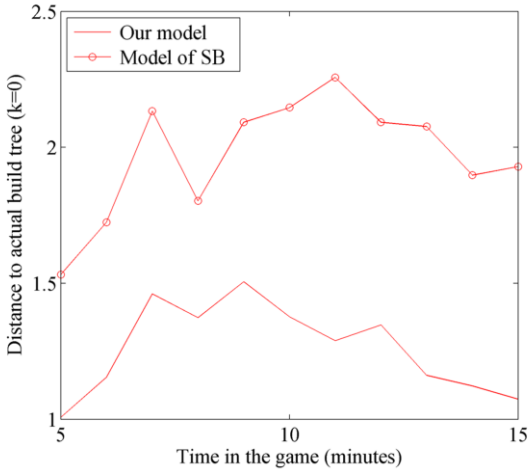

Figure 5. Distance (d for $k=0$) in an interval between 5th up to 15th minute for PvP match-up with 30% noise for out model and the model of Synnaeve-Bessiere[21].

Our experiment shows that the first attack occurs around the 7th minute and the last attack occurring before the 15th minute happens about the 12th minute. When a player is attacked, s/he is forced to take defensive strategies. Hence, to counter the attacks, s/he may construct some buildings differing from his/her original strategy.

In other words, his/her build tree will not follow his/her original strategy. Thus prediction of the correct build tree would become significantly harder. Consequently, the distance to the actual build tree prediction would increase in the interval of 7th to 12th minute.

In the second experiment, we calculate the number of buildings that our model can predict in advance at a maximum distance of 1. We compute k for $d \leq 1$ at the 15th minute to consider the longest build tree among all time intervals and compare it with the k value proposed by Synnaeve and Bessiere [21]. The comparison is reported in figure 6.

As illustrated in figure 6, the noise has a very low impact on the k value for the two models. However, the curves in our method are smoother, which means that the noise has a lower impact on our model and it is robust to the noise because we choose the correct set of time parameters. (We explain the parameter selection in Section 4.3.)

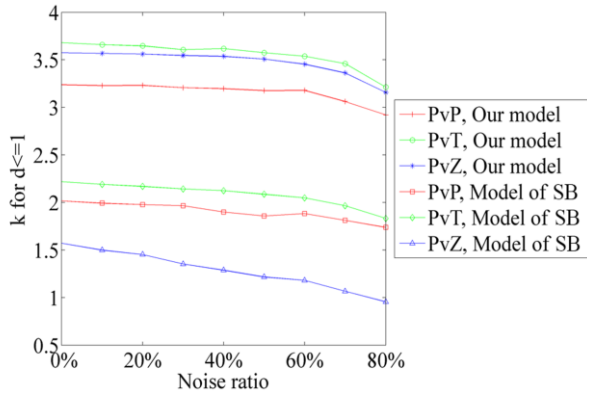

Figure 6. Number of buildings that our model and the model of Synnaeve and Bessiere (SB) [21] can predict in advance at a maximum distance of 1 (k for $d \leq 1$) for different match-ups.

It should be stressed that the space between different match-up curves of our model is much lower than that of [21]. This means that our model is more general and has a similar result in the various game plays. Moreover, despite the 80% noise in observation, our model can predict about 3 buildings in advance, whereas the average number of the buildings that the model of [21] can predict ahead is about 1.5.

It is shown that 30% random noise is required for the real setup/competitive games [21]. Thus comparison of the two models in 30% random noise could be considered as their proxy of performance in real game setups.

Our build tree prediction technique could be applied to all games with technology tree. Some RTS games have the notion of technology tree. Hence, our technique is applicable to all of them.

4.3. Parameter determination

In order to complete the model description, we design two experiments to determine the time parameter. As mentioned in Section 3, different build trees should be compared in more than one point in the construction time in order to increase the accuracy of the model. In this section, we perform two experiments to determine how many points in time are required to achieve better results, and to indicate the locations of these points.

The candidate locations are four quartiles of all buildings' construction times, namely Q1, Q2 (median), Q3, and Q4 (last building's construction time). We run these two experiments on PvP match-ups in three different game times: 5^{th}, 10^{th}, and 15^{th} minutes. It is shown that 30% random noise is required for the real setup/competitive games [21]. Thus we add 30% noise to the observation variable to simulate the fog of war for real games. We select Q4 as a fixed point in time and choose between the other three points.

In the first experiment, we determine how many extra points are required. In order to evaluate the performance, we compute d for $k=0$. The results obtained are shown in table 1.

Table 1. Average value of d ($k=0$) for different numbers of points in time.

Time point location	5^{th} minute	10^{th} minute	15^{th} minute	Average
Q3,Q4	1.007	1.377	1.074	**1.153**
Q2,Q3,Q4	1.146	1.349	1.152	1.216
Q1,Q2,Q3,Q4	1.384	1.554	1.384	1.441

This table shows that adding more than one extra point in time increases the distance to the actual build tree. Thus we can conclude that only two points in time are required to achieve the best results.

In the second experiment, we identify the location of one extra point. In order to evaluate the performance, we compute d for $k=0$. The results obtained are shown in table 2.

Table 2. Average value of d ($k=0$) for different locations of extra point in time.

Time point location	5^{th} minute	10^{th} minute	15^{th} minute	Average
Q1,Q4	1.400	1.596	1.391	1.462
Q2,Q4	1.127	1.298	1.105	1.177
Q3,Q4	1.007	1.377	1.074	**1.153**

These results indicate that the third quartile is the best choice, thus we compare different build trees in Q3 and Q4 to achieve the best results.

5. Conclusion

In this paper, we proposed an opponent model based on the Bayesian network for the StarCraft game. The technology tree used by the opponent defines his/her strategy. Thus we used a variation of the technology tree, namely the build tree, as his/her model. In order to construct this model, all the possible build trees were gathered from game logs and the frequency of its values was computed. The most probable build tree, which is the superset of the observed buildings, was chosen based on these frequency values, the last building's construction time, and the third quartile

of all buildings' construction times. The opponent's build tree can be used to predict his/her strategy [7, 9, 10]. The proposed method of build tree prediction can be applied to all games having a technology tree. It means that it is applicable to RTS games where the concept of "build order" exists. For example, the Command & Conquer series and the Total Annihilation games can be named. For all RTS games with a technology tree, the structure of the Bayesian network and its variables are the same. The differences are about the existing types of buildings in different games. Hence, for each game, the variables of the Bayesian network should be re-learned from the relevant data of the game.

The number of time points in which different build trees were compared is an important parameter of the model. In order to determine this number and the locations of time points, we performed two experiments, which indicated that a combination of Q3 and Q4 was the best choice.

We compared the proposed method with the Synnaeve and Bessiere's [21] model. The experimental results showed that even in the presence of 80% random noise in building observations, our model could predict 3 buildings ahead.

As mentioned in Section 3.4, our future work will be focused on fine-tuning the distribution of *P(EndTime|BuildTree)* and *P(Q3Time|BuildTree)*. We will try to replace the estimation of these two distributions by more precise ones.

References

[1] Robertson, G. & Watson, I. (2014). A review of real-time strategy game AI. AI Magazine, vol. 35, no. 4, pp. 75-104.

[2] Ontanon, S., et al. (2013). A survey of real-time strategy game AI research and competition in StarCraft. IEEE Transactions on Computational Intelligence and AI in Games, vol. 5, no. 4, pp. 293-311.

[3] Buro, M. (2003). Real-time strategy games: a new AI research challenge. Proc. 18th Int. Joint Conf. Artificial Intelligence, Acapulco, Mexico, pp. 1534-1535.

[4] Buro, M. & Furtak., T. (2004). RTS games and real-time AI research. Proc. Behavior Representation in Modeling and Simulation Conf. (BRIMS), Arlington, Virginia, pp. 51-58.

[5] Borghetti, B. J. (2008). Opponent Modeling in interesting adversarial environments, Ph.D. dissertation, Department of Computer Science, Minnesota University, Minneapolis.

[6] Bakkes, S. C. J., et al. (2009). Opponent modelling for case-based adaptive game AI. Entertainment Computing, vol. 1, no. 1, pp. 27-37.

[7] Synnaeve, G. (2012). Bayesian Programming and Learning for Multi-Player Video Games, Ph.D. dissertation, Department of Computer Science, Grenoble University, Grenoble, France.

[8] Fjell, M. S. & Mllersen, S. V. (2012). Opponent modeling and strategic reasoning in the real-time strategy game StarCraft, M.S. thesis, Department of Computer Science and Information Science, Norwegian University of Science and Technology, Trondheim, Norway.

[9] Synnaeve, G. & Bessiere, P. (2011). A Bayesian model for opening prediction in RTS games with application to StarCraft. IEEE Conf. on Computational Intelligence and Games (CIG), Seoul, South Korea, pp. 281 - 288.

[10] Synnaeve, G. & Bessiere, P. (2016). Multi-scale Bayesian modeling for RTS games: an application to StarCraft AI. IEEE Transactions on Computational Intelligence and AI in Games, vol. 8, no. 4, pp. 338-350.

[11] van den Herik, H. J., et al. (2005). Opponent modelling and commercial games. Proc. IEEE Sym. on Computational Intelligence and Games (CIG), Colchester, Essex, pp. 15-25.

[12] Anantharaman, T. (1997). Evaluation tuning for computer chess: Linear discriminant methods. International Computer Games Association, vol. 20, no. 4, pp. 224-242.

[13] Weber, B. G. & Mateas., M. (2009). A data mining approach to strategy prediction. IEEE Conf. on Computational Intelligence and Games (CIG), Milano, Italy, pp. 140–147.

[14] Larik, A. & Haider, S. (2015). Opponent classification in robot Soccer, In: Moonis, A., et al. (Eds.), Current Approaches in Applied Artificial Intelligence. Lecture Notes in Computer Science. vol. 9101. Springer International Publishing, Cham, pp. 478-487.

[15] Schadd, F., et al. (2007). Opponent modeling in real-time strategy games. Proc. GAME-ON, Bologna, Italy, pp. 61-68.

[16] Hamidzadeh, J. (2015). IRDDS: Instance reduction based on Distance-based decision surface. Journal of AI and Data Mining, vol. 3, no. 2, pp. 121-130.

[17] Frandsen, F., et al. (2010). Predicting player strategies in real time strategy games, M.S. thesis, Department of Computer Science, Aalborg University, Aalborg, Denmark.

[18] Palero, F., et al. (2015). Online gamers classification using k-means, In: Camacho, D., et al. (Eds.), Intelligent Distributed Computing VIII. Studies

in Computational Intelligence. vol. 570. Springer International Publishing, Cham, pp. 201-208.

[19] Spronck, P., et al. (2006). Adaptive game AI with dynamic scripting Machine Learning, Special Issue on Machine Learning in Games, vol. 63, no. 3, pp. 217–248.

[20] Machado, M. C., et al. (2011). Player modeling: Towards a common taxonomy. 16th Int. Conf. on Computer Games (CGAMES) Louisville, KY, USA, pp. 50-57.

[21] Synnaeve, G. & Bessiere., P. (2011). A Bayesian model for plan recognition in RTS games applied to StarCraft. Proc. 7th Artificial Intelligence and Interactive Digital Entertainment Conf., Palo Alto CA, USA, pp. 79–84.

[22] Dereszynski, E., et al. (2011). Learning probabilistic behavior models in real-time strategy games. Proc. 7th AAAI Conf. Artificial Intelligence and Interactive Digital Entertainment, Stanford, California, USA, pp. 20-25.

[23] Farouk, G. M., et al. (2013). Generic opponent modelling approach for real time strategy games. 8th Int. Conf. on Computer Engineering & Systems (ICCES), Cairo, Egypt, pp. 21-27.

[24] Safadi, F., et al. (2015). Artificial intelligence in video games: towards a unified framework. International Journal of Computer Games Technology, vol. 2015, pp. 1-30.

[25] Richoux, F., et al. (2016). GHOST: A combinatorial optimization framework for rts-related problems. IEEE Transactions on Computational Intelligence and AI in Games, vol. 8, no. 4, pp. 377 - 388.

[26] Stanescu, M. & Certicky, M. (2016). Predicting opponent's production in real-time strategy games with answer set programming. IEEE Transactions on Computational Intelligence and AI in Games, vol. 8, no. 1, pp. 89-94.

[27] Synnaeve, G. & Bessiere, P. (2012). A dataset for StarCraft AI & an example of armies clustering. AIIDE Workshop on AI in Adversarial Real-time games, Palo Alto, United States.

نشریه هوش مصنوعی و داده کاوی

روش مقاوم مدل‌سازی حریف در بازی های استراتژیک بی‌درنگ

آرزو ترکمن و رضا صفابخش*

دانشکده مهندسی کامپیوتر و فناوری اطلاعات، دانشگاه صنعتی امیرکبیر، تهران، ایران.

ارسال ۲۰۱۸/۰۴/۰۳؛ بازنگری ۲۰۱۸/۰۵/۱۲؛ پذیرش ۲۰۱۸/۰۷/۰۳

چکیده:

مدل‌سازی حریف یکی از چالش‌های بازی‌های استراتژیک بی‌درنگ (RTS) است زیرا محیط این بازی‌ها خصمانه است و بازیکن قادر نیست اعمال بعدی حریفش را پیش‌بینی نماید. علاوه بر این، محیط این بازی‌ها به دلیل مه جنگی کاملاً قابل مشاهده نیست. در این مقاله یک مدل حریف ارائه می‌شود که در مقابل نویز حاصل از مه جنگی مقاوم است. به منظور مقابله با عدم قطعیت موجود در این بازی‌ها، یک شبکه بیزین طراحی شده است که متغیرهایش از روی پایگاه داده مربوط به رویدادهای ضبط شده بازی محاسبه می‌شود بنابراین نیازی به دانش انسانی ندارد. روش ارائه شده بر روی بازی استارکرفت مورد ارزیابی قرار خواهد گرفت. عملکرد مدل ارائه شده با روش سیناوی و بزیر مقایسه شده است. نتایج آزمایشات نشان می‌دهد روش ارائه شده اعمال بعدی حریف را با دقت بیشتری پیش‌بینی می‌نماید. با استفاده از این مدل می توان یک الگوریتم هوش بازی تطبیقی طراحی کرد که در همه بازی‌های RTS که مفهوم ترتیب ساخت در آن‌ها وجود دارد کارایی دارد.

کلمات کلیدی: شبکه بیزین، مدل‌سازی حریف، بازی‌های استراتژیک بی‌درنگ، استارکرفت.