International Journal of Intelligent Computing and Information Sciences

OPPONENT MODELS PREPROCESSING IN REAL-TIME STRATEGY GAMES

M. A. Mourad, M. M. Aref and M. H. Abd-Elaziz

Computer Science and Basic Sciences Department, Faculty of Computer and Information Sciences, Ain Shams University, Cairo, Egypt.
Mourad.Aly@cis.asu.edu.eg, Mostafa.Aref@cis.asu.edu.eg, MHAziz@cis.asu.edu.eg

Abstract: *Creating a human-like computer player in real-time strategy games requires huge number of opponent models, these models must be preprocessed to either focus on accuracy or performance according to our needs. In order to preprocess these models accurately, we need to detect their type. Opponent models' type can be complex or simple. Complex opponent models are low variance models whose differences in features' values are low, so in order to accurately separate between these models, we need to preprocess them by increasing their dimensions. Simple opponent models are high variance models whose differences in features' values are high, so in order to separate between these models in a reasonable time, we need to preprocess them to decrease their dimensions, if possible, without accuracy or data loss.*

Keywords: *Opponent modeling, adaboost, rough sets, clustering, real-time strategy games, rts*

1. Introduction

Opponent modeling in real-time strategy (RTS) games has a significant interest to the AI community. Robust opponent models could improve automated agents, for example by augmenting the strategy representations used in some architectures or guiding the Monte-Carlo simulations of an opponent. They could be incorporated into intelligent systems and they could be assistants to help human players reason about the state of the game and predict an opponent's future actions. They could also be used in the analysis of game play, to automatically identify common strategic elements or discover novel strategies as they emerge. Achieving victory in RTS games depends on selecting a suitable plan (set of actions), selecting a suitable plan depends on building an imagination (building a model) of the opponent to know how to deal with. This imagination is the opponent model, the stronger the opponent modeling process is, the more accurate the selected suitable plan is and consequently the higher probability achieving the victory is. Our methodology includes two steps, the first step is to detect the opponent models' type and the second step is to generate another data set based on the type detected. The generated data set is a preprocessed version of the opponent models which is used, later on, in the classification process for training and testing. The output of the classification process is the opponent's strategy. By "strategy" we mean a player's choice of units and structures to build, which dictates the tone of the game. Our models are learned from collections of replay files [1].

This paper is organized as follows. Section 2 covers the background of RTS games and opponent modeling, section 3 proposes our approach, section 4 presents a Star Craft 2 case study and section 5 concludes our approach.

1.1 Background

1.1.1 Real-time strategy games

Real-time strategy (RTS) games are strategic war games where two or more players operate on a virtual battlefield, controlling resources, buildings, units and technologies to achieve victory by destroying others. In an RTS game, players control many units and structures by issuing orders from an overhead perspective in real-time in order to gather resources, build an infrastructure and an army, and destroy the opposing player's forces. The real-time aspect comes from the fact that players do not take turns, but instead may perform as many actions as they are physically able to make, while the game simulation runs at a constant frame rate (24 frames per second in Star Craft) to approximate a continuous flow of time. Some notable RTS games include Dune II, Total Annihilation, Warcraft, Command & Conquer, Age of Empires, and Star Craft series.

Generally, each match in an RTS game involves two players starting with a few units and/or structures in different locations on a two-dimensional terrain (map). Nearby resources can be gathered in order to produce additional units and structures and purchase upgrades, thus gaining access to more advanced in-game technology (units, structures, and upgrades). Additional resources and strategically important points are spread around the map, forcing players to spread out their units and buildings in order to attack or defend these positions. Visibility is usually limited to a small area around player-owned units, limiting information and forcing players to conduct reconnaissance in order to respond effectively to their opponents. In most RTS games, a match ends when one player (or team) destroys all buildings belonging to the opponent player (or team), although often a player will forfeit earlier when they see they cannot win [2].

1.1.2 Opponent modeling

An important factor that influences the choice of strategy is the strategy of the opponent. If one knows what types of units the opponent has, then typically one would choose to build units that are strong versus those from the opponent. A method of representing information of the enemy is known as opponent modeling.

Opponent modeling problems can often be seen as a classification problem, where data that is collected during the game is classified as one of the available opponent models. Therefore it is possible to apply standard machine-learning techniques. However, a limiting condition is the fact that these calculations have to be performed in real-time, while many other computations, like the rendering of the game graphics, have to be performed simultaneously. This limits the amount of available computing resources. As long as the opponent model is robust, classification process and the selected plan, in consequence, will be accurate [3].

2. Approach

Feature selection process is the construction of the opponent model schema. In this paper features are selected from the RTS game itself. Gathered data set might be in a binary format that is not useful to work with, it might need to be decoded and filtered. Decoded data set might be distributed into spaces where each space represents a player type or race.

Distributed data set is then labelled by an expert or clustering algorithms, then they are divided into training data set and testing data set.Training data type of each race is determined using the following equations. Equation "(1)" is used to get means of every class, equation "(2)" is used to get the euclidean distance between all means of all classes, then finally type is determined by selecting the minimum euclidean distance and compare it with a specific threshold. If the minimum euclidean distance is less than the threshold, then the training data type of this race is complex, otherwise it is simple. The determined type is data-dependent and game-dependent as well. Each race in each game has it's own data set, and each data set will have a certain type.

$$\mu i = \frac{\sum_{j=1}^{n} Fij}{n} \quad (1)$$

where i is the feature number, j is the sample number and n is the number of samples

$$d(\mu 1, \mu 2) = \sqrt{\sum_{i=1}^{n} (\mu 1i - \mu 2i)^2} \quad (2)$$

where i is the feature number and n is the number of features

min(d1,d2) < threshold complex opponent models

min(d1,d2) > threshold simple opponent models

Complex opponent models are low variance models whose differences in their features' values are low, so in order to accurately separate between these models, we need to preprocess them to increase their dimensions by using AdaBoost. Simple opponent models are high variance models whose differences their features' values are high, so in order to separate between these models in a reasonable time, we need to preprocess them to decrease their dimensions, if possible, without accuracy or data loss by using Rough Sets. Preprocessing is done by AdaBoost to increase training data set dimensionality or by Rough Sets to decrease their dimensionality if possible. Finally, testing data set is classified with the preprocessed data either coming from AdaBoost or Rough Sets as shown in figure 1.

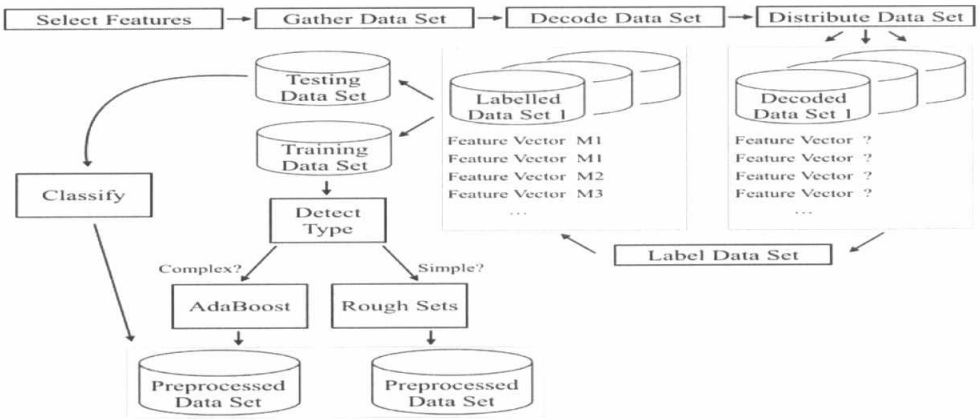

Figure 1. Opponent models preprocessing

3. Case study - Star craft 2

Star Craft is a canonical RTS game, like chess is to board games, with a huge player base and numerous professional competitions. The game has three different but very well balanced teams, or "races", allowing for varied strategies and tactics without any dominant strategy, and requires both strategic and tactical decision-making roughly equally. These features give Star Craft an advantage over other RTS games which are used for AI research, such as Wargus2 and ORTS. There are a large number of Star Craft replays (game logs) available on the internet which can be used for data mining, and there are many players of all skill levels to test against [2].

Select features

Features selected are Star Craft 2 units, buildings and upgrades. Figure 2, and 3 show our selected features.

Protoss	Terran	Zerg
Probe	SCV	Larva
Zealot	Marine	Drone
Sentry	Reaper	Overlord
Stalker	Marauder	Zergling
High Templar	Ghost	Baneling
Dark Templar	Hellion	Roach
Archon	Widow Mine	Queen
Phoenix	Siege Tank	Hydralisk
Oracle	Hellbat	Mutalisk
Void Ray	Thor	Corruptor
Tempest	Medivac	Infestor
Carrier	Viking	Swarmhost
Observer	Raven	Ultralisk
Warp Prism	Banshee	Viper
Immortal	Battle Cruiser	Brood Lord
Colossus		Overseer
Mothership Core		Nydus Worm
Mothership		

Figure 2. Unit features

Protoss	Terran	Zerg
Nexus	Command Center	Hatchery
Pylon	Orbital Command	Extractor
Assimilator	Planetary Fortress	Spawning Pool
Gateway	Supply Depot	Evolution Chamber
Wrap Gate	Refinery	Spore Crawler
Cybernetics Core	Barracks	Spine Crawler
Forge	Engineer Bay	Roach Warren
Photon Cannon	Missile Turret	Baneling Nest
Twilight Council	Bunker	Liar
Stargate	Sensor Tower	Spire
Robotics Facility	Factory	Hydralisk Den
Templar Archives	Armory	Infestation Pit
Dark Shrine	Starport	Nydus Network
Robotics Bay	Fusion Core	Hive
Fleet Beacon	Ghost Academy	Ultralisk Cavern
	Reactor	Greater Spire

Figure 3. Building features

Gather data set

Data set is extracted from Star Craft 2 replays files. Example of these packages are matches of season 2 of global Star Craft 2 league (GSL), world championship series (WCS) America and Europe Matches. GSL includes 328 replay files, WCS America includes 136 replay files and WCS Europe includes 242 replay files. Total number of replay files is 706, each replay file has 2 players so the total number of feature vectors of this season is 1412.

Decode the gathered data set

Start Craft 2 replay files aren't human readable. In order to fetch features from replay files, they must be decoded. There are many ways to decode the replay files, one of them is s2protocol, an open source replay files parser written in python by blizzard entertainment (owner of Star Craft) which translates replay files into useful data. The output of s2protocol might not be suitable for direct use, output might need to be processed in order to be used. Features are grouped by races, each Star Craft 2 race has it's own features (it's own units and buildings).

A Star Craft 2 replay contains many events, we're interested in UnitInitEvent, UnitDoneEvent and UnitBornEvent. UnitInitEvent and UnitDoneEvent specify the start and finish of a building construction, UnitDoneEvent is used because if the building isn't finished for any reasons we can't include it in our features. UnitBornEvent specify the born of a unit. Each event is a dictionary, with the field _event containing the prefix NNet.Replay.Tracker.S followed by the relevant event name. Not all game events are directly represented, and have to be determined by the parsing program, while some events are in two parts and their id, called m_unitTagIndex, needs to be kept track of to calculate the full game event. Timing information can be extracted for Hidden Markov Model to be integrated with our system [4].

S2protocol decodes replay files into python objects, we have converted these python objects to java-script object notation (JSON) objects in order to map it to models in our system, a sample of our JSON objects is shown in figure 4. We also have removed any unnecessary data from the python objects to enhance the performance.


```
{"_bits": 304,
"_event": "NNet.Replay.Tracker.SUnitBornEvent",
"_eventid": 1,
"_gameloop": 11983,
"m_controlPlayerId": 2,
"m_unitTagIndex": 436,
"m_unitTagRecycle": 2,
"m_unitTypeName": "VoidRay",
"m_upkeepPlayerId": 2,
"m_x": 23,
"m_y": 145}
```

```
{"_bits": 288,
"_event": "NNet.Replay.Tracker.SUnitInitEvent",
"_eventid": 6,
"_gameloop": 12016,
"m_controlPlayerId": 2,
"m_unitTagIndex": 433,
"m_unitTagRecycle": 2,
"m_unitTypeName": "Pylon",
"m_upkeepPlayerId": 2,
"m_x": 31,
"m_y": 125}
```

```
{"_bits": 120,
"_event": "NNet.Replay.Tracker.SUnitDoneEvent",
"_eventid": 7,
"_gameloop": 12416,
"m_unitTagIndex": 433,
"m_unitTagRecycle": 2}
```


Figure 4. Sample of our JSON events of interest

Convert the decoded data set into opponent models

Each reply file event is converted into 2 opponent models, opponent model for player 1 and opponent model for player 2. Table 1 shows an opponent model example.

Distribute opponent models

Opponent models are distributed into 3 separated spaces. Space for protoss race, space for terran race and space for zerg race. Our data set includes 545 protoss opponent models, 309 terran opponent models and 546 zerg opponent models as shown in table 2.

Label opponent models

Opponent models of each race are labeled with K-Means algorithm with k = 5 clusters per each race. Labels are shown in table 3. Fast and accurate k-means, and fast expectation-maximization algorithms can be also applied [5][6

Divide opponent models

Opponent models data set is divided into training data set and testing data set.

Detect opponent model type

Training data set type, whether they are simple (easily separable) or complex (interleaved), is determined using equation "(1)" and "(2)" as stated in section 3. Table 4 shows a sample of protoss mean feature vector and table 5 shows euclidean distances between all means of protoss and zerg races. Linear perceptron and projection of positive points on subspaces can be applied [7][8].

Table 1. Protoss feature vector

Feature\Race	Protoss	Feature\Race	Protoss
Probe	55	MothershipCore	1
Zealot	21	Mothership	0
Sentry	0	Nexus	5
Stalker	62	Pylon	22
HighTemplar	9	Assimilator	6
DarkTemplar	0	Gateway	4
Archon	0	WarpGate	0
Phoenix	5	CybernеticsCore	1
Oracle	1	Forge	1
VoidRay	0	PhotonCannon	4
Tempest	0	TwilightCouncil	1
Carrier	0	Stargate	1
Observer	6	RoboticsFacility	2
WarpPrism	0	TemplarArchives	0
Immortal	11	DarkShrine	0
Colossus	6	RoboticsBay	1

Table 2. Distributed opponent models

Protoss R = 0	Terran R = 1	Zerg R = 2
545	309	546

Table 3. Distributed labeled opponent models

Label\Race	Protoss R = 0	Terran R = 1	Zerg R = 2
Model 0 R	44	113	155
Model 1 R	201	57	87
Model 2 R	39	20	29
Model 3 R	141	36	64
Model 4 R	120	83	211

Preprocessing using AdaBoost

AdaBoost is used for complex opponent models to increase their dimensionality in order to increase their variance for better accuracy. Protoss and terran are preprossed with AdaBoost to increase their dimensionality. AdaBoost implemented in OpenCV can only learn data set with 2 labels, so in order to learn data set with more than 2 labels, MultiBoost must be used [9].

Preprocessing using Rough Sets

Rough Sets are used for simple opponent models to decrease their dimensionality, if possible, in order to separate between them in a reasonable time for better performance. Zerg is processed with Rough Sets to decrease their dimensionality without data loss. Rough Sets implementation in c++ can be found in rosetta c++ library. Preprocessed data, either from AdaBoost or Rough Sets are the input of the classification process [10].

4. Conclusion

In this paper, we have proposed a methodology to preprocess opponent models so that their classification process can be executed accurately and in a reasonable time. Our approach isn't game-specific, it doesn't depend on any type of RTS games. Different RTS games will have different data sets, these data sets will have different races, and these races might be all simple (easily separable) or all complex (interleaved) or have a combination of both simple and complex opponent models. Our approach preprocesses opponent models according to their detected type so that the classification process can use the preprocessed opponent models for training and testing.

Table 4. Protoss mean feature vector

Feature\Race	Protoss	Feature\Race	Protoss
Probe	72.358971	MothershipCore	1.435897
Zealot	32.717949	Mothership	0
Sentry	8.230769	Nexus	3.871795
Stalker	68.128204	Pylon	24.153847
HighTemplar	4.461538	Assimilator	6.256410
DarkTemplar	1.923077	Gateway	9.205129
Archon	1.435897	WarpGate	0
Phoenix	8.666667	CyberneticsCore	0.974359
Oracle	0.307692	Forge	1.051282
VoidRay	2.256410	PhotonCannon	6.743590
Tempest	0.358974	TwilightCouncil	0.948718
Carrier	0	Stargate	1.025641
Observer	3.282051	RoboticsFacility	1.205128
WarpPrism	0.743590	TemplarArchives	0
Immortal	1.897436	DarkShrine	0.256410
Colossus	5.128205	RoboticsBay	0.769231

Table 5. Protoss (column 1) and Zerg (column 2) mean euclidean distances

36	285
70	183
131	611
82	328
38	264
102	364
55	390
64	625
57	159
93	737

With threshold 100, protoss and terran are complex, while zerg is simple.

References

1. Ethan Dereszynski, Jesse Hostetler, Alan Fern, Tom Dietterich Thao-Trang Hoang and Mark Udarbe, "Learning Probabilistic Behavior Models in Real-time Strategy Games", 2011.
2. Glen Robertson and Ian Watson, "A Review of Real-Time Strategy Game AI", 2014.
3. Magnus Sellereite Fjell and Stian Veum Mollersen, "Opponent modeling and strategic reasoning in real-time strategy games", 2012.
4. Christian Jonassen, "Gamification and its applications to systematic improvement in performing complex tasks", 2014.
5. Michael Schinder, Alex Wong and Adam Meyerson, "Fast and accurate k-means for Large datasets", 2011.
6. Zhiwen Yu and Hausan Wong, "A fast expectation maximization algorithm based on grid and PCA", 2006.
7. Frank Rosenblatt, "The perceptron algorithm", 1957.
8. Yogananda A P, M Narasimha Murthy and Lakshmi Gopal, "A fast linear separability test by projection of positive points on subspaces", 2007.
9. Djalel Benbouzid, Robert Busa-Fekete, Norman Casagrande, Francois David Collin and Balazs Kegl, "MultiBoost: A Multi-purpose Boosting Package", 2012.
10. Walid Moudani, Ahmad Shahin, Fadi Chakik and Felix Mora-Camino, "Dynamic rough sets features reduction", 2011.