Proceedings, The Eleventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-15)

Automatic Learning of Combat Models for RTS Games

Alberto Uriarte and **Santiago Ontañón**
Computer Science Department
Drexel University
{albertouri,santi}@cs.drexel.edu

Abstract

Game tree search algorithms, such as Monte Carlo Tree Search (MCTS), require access to a forward model (or "simulator") of the game at hand. However, in some games such forward model is not readily available. In this paper we address the problem of automatically learning forward models (more specifically, combats models) for two-player attrition games. We report experiments comparing several approaches to learn such combat model from replay data to models generated by hand. We use StarCraft, a Real-Time Strategy (RTS) game, as our application domain. Specifically, we use a large collection of already collected replays, and focus on learning a combat model for tactical combats.

Introduction

A significant number of different artificial intelligence (AI) algorithms that play Real-Time Strategy (RTS) games, like Monte Carlo Tree Search (MCTS) (Browne et al. 2012) or Q-Learning (Jaidee and Muñoz-Avila 2012), assume the existence of a forward model that allows predicting the state that will be reached after executing a certain action in the current game state. While this assumption is reasonable in certain domains, such as Chess or Go where simulating the effect of actions is trivial, forward models are not readily available in other domains where precise descriptions of the effect of actions is not available.

In this paper we study how to automatically learn forward models for RTS games from game replay data. We argue that while forward models might not be available, logs of previous games might be available, from where the result of applying specific actions in certain situations can be observed. This is the case in most Real-Time Strategy (RTS) games. For example, consider the StarCraft game, where precise definitions of the effects of unit actions is not available, but large collections of replays are available. Automatically acquiring forward models from observation is of key importance to RTS game AI, since it would allow the application of game tree search algorithms, such as MCTS, to real-world domains, for which, forward models are, obviously, not available.

Copyright © 2015, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Specifically, in this paper, we focus on learning forward models for a subset of a full RTS game: combat situations. We use StarCraft, a popular RTS game, as our testbed, and exploit the large collection of readily available replays to extract a collection of combat situations and their results. We use this data to train a combat model (or "simulator") to predict the outcomes of combat situations. In order to learn the forward model, we model a combat situation as an *attrition game* (Furtak and Buro 2010). An attrition game is a combat simulation game where individual units cannot move, and only their damage and hit points are considered. Thus, our approach is based upon learning the parameters of the attrition game from replay data, and use this to simulate the evolution of a given combat situation over time.

The remainder of this paper is organized as follows. First we provide background on combat models in RTS games and their applications. Then we propose a high-level abstraction representation of a combat state and two combat models using this abstraction. After that, we explain how to extract combat situations from replay data and how to train our combat model to simulate combats. Finally, we present our experiments in forwarding the state using our proposed simulators, and compare there against existing hand-made state-of-the-art simulators.

Background

Real-Time Strategy (RTS) games in general, and StarCraft in particular, have emerged as a fruitful testbed for new AI algorithms (Buro 2003; Ontañón et al. 2013). One of the most recurrent techniques for tactical decisions are those based on game tree search, like alpha-beta search (Churchill, Saffidine, and Buro 2012) or MCTS (Balla and Fern 2009; Churchill and Buro 2013; Uriarte and Ontañón 2014; Justesen et al. 2014).

Of particular interest to this paper is the MCTS family of algorithms (Browne et al. 2012), which build a partial game-tree in an incremental and asymmetric manner. At each iteration, the algorithm selects a leaf of the current search tree using a *tree policy* and expands it. This *tree policy* is used to balance the *exploration* (look in areas of the tree that have not been sufficiently explored yet) and *exploitation* (confirm that the most promising areas of the tree are indeed promising). Then, a *default policy* is used to simulate a game from the selected leaf, until a terminal node is reached. The outcome of this simulation (a.k.a. *playout* or *rollout*) is then used to update the expected utility of the corresponding leaf in the tree. In order to generate the tree and to perform these playouts, MCTS requires a *forward model* that given a state and an action, predicts which will be the resulting state after executing the action. The long term goal of the research presented in this paper is to allow the application of MCTS and other game tree search techniques to domains where no such forward model is available.

This is not the first attempt to create a combat model for RTS games. Balla and Fern (2009) used a hand-crafted simulator in order to deploy UCT (a variant of MCTS) in the *Warcraft II* RTS game. Churchill and Buro (2013) developed SparCraft, a low level StarCraft combat simulator. SparCraft was developed using a large human effort observing the behavior of different StarCraft units frame by frame. Despite being extremely accurate for small combat scenarios, SparCraft does not cover all situations (like collisions) nor units (like spell casters or dropships), due to the tremendous amount of effort that it would take to model the complete StarCraft game. Uriarte and Ontañón (2014) defined a simplified model were each squad deals their maximum DPF (Damage Per Frame) until one army is destroyed to apply MCTS to StarCraft. Soemers (2014) proposed another model based on Lanchester's Square Law were each individual unit is killed over time during a battle also to apply MCTS to StarCraft. Finally, Stanescu et al. (2013) used SparCraft to predict the outcome of a combat, but only focusing on which player will be the winner instead of the exact outcome of the battle.

High-level Abstraction in RTS Games

The proposed approach does not simulate the low-level, pixel-by-pixel movement of units in a RTS game, but rather the high-level outcome of a combat scenario. Thus, we will use the abstraction described in (Uriarte and Ontañón 2014), which we describe below:

- An RTS map is modeled as a graph M where each node is a region, and edges represent paths between region. In the experiments presented in this paper, we employed Perkin's algorithm (Perkins 2010) to transform StarCraft maps into this representation.
- Instead of modeling each unit individually, we consider *groups of units*, where a group is a 4-tuple $g = \langle player, type, size, loc \rangle$ with the following information:
  - Which *player* controls this group.
  - The *type* of units in this group (e.g., *marines*).
  - Number of units forming this group (*size*).
  - The position in the map (*loc*), which corresponds to which region in the graph M the group is located.

Notice that we do not record the hit points or shield of the units.. Additionally, we assume that all combats happen *inside* one of the regions, and never across multiple regions. Thus, we will drop *loc* from the group representation in the remainder of this paper (since all groups in a given combat have the same value for *loc*). As a result, our forward model works as follows:

Figure 1: StarCraft combat situation with two players.

Table 1: Groups in the high-level abstraction of Figure 1.

group	*Player*	*Type*	*Size*
g_1	red	Worker	1
g_2	red	Marine	2
g_3	red	Tank	3
g_4	blue	Worker	2
g_5	blue	Marine	4
g_6	blue	Tank	1

Input: A set of groups $G = \{g_1, ...g_n\}$ (the initial state of the combat).

Output: A set of groups G' (the final state of the combat), and the length t of the combat (in game time).

We require that all groups in G' belong to the same $player$, in other words, only one army stands. Figure 1 shows a combat situations and Table 1 its corresponding high-level representation.

Learning a Combat Forward Model

Many variables, such as weapon damage of a unit or the cool down of a weapon are involved in the dynamics of combats in RTS games like StarCraft. Moreover, other factors such as maneuverability of units, or how the special characteristics of a given unit type makes them more or less effective against other types of units or combinations of units are harder to quantify. The following subsections propose two approaches to simulate combats based on modeling the way units interact in two different ways.

Sustained DPF Model ($simDPF_{sus}$)

$simDPF_{sus}$ is the simplest model we propose and assumes that the amount of damage a player can do does not decrease over time. Given the initial state G, where groups belong to players A and B, the model proceeds as follows:

1. First, the models computes how much time each army needs to destroy the other. In some RTS games, such as StarCraft, units might have a different DPF (*damage per frame*) when attacking to different types of units (e.g., air vs land units), and some units might not even be able to attack certain other units (e.g., walking swordsmen cannot attack a flying dragon). Thus, for a given player, we can compute her DPF_{air} (the aggregated DPF that units

of a player that can attack only air units), DPF_{ground} (DPF that the player can perform only to ground units) and DPF_{both} (aggregated DPF of the units that can attack both ground and air). After that, we can compute the time required to destroy all air and land units separately:

$$t_{air}(A,B) = \frac{HP_{air}(A)}{DPF_{air}(B)}$$
$$t_{ground}(A,B) = \frac{HP_{ground}(A)}{DPF_{ground}(B)}$$

where $HP(A)$ is the sum of the hit points of the units in all the groups of player A. Then, we compute which type of units (air or ground) would take longer to destroy, and DPF_{both} is assigned to that type. For instance, if the air units take more time to kill we recalculate t_{air} as:

$$t_{air}(A,B) = \frac{HP_{air}(A)}{DPF_{air}(B) + DPF_{both}(B)}$$

And finally we compute the global time to kill the other army:

$$t(A,B) = max(t_{air}(A,B), t_{ground}(A,B))$$

2. Then, the combat time t is computed as:

$$t = min(t(A,B), t(B,A))$$

3. After that, the model computes which and how many units does each player have time to destroy of the other player in time t. For this purpose, this model takes as input a *target selection* policy, which determines the order in which a player will attack the units in the groups of the other player. The final state G' is defined as all the units that were not destroyed.

$simDPF_{sus}$ has three input parameters: the DPF of each unit type to each other unit type, the maximum hit points of each unit type, and a target selection policy. Later in this paper we will propose different ways in which these three input parameters can be defined or learned from data.

Decreased DPF Model ($simDPF_{dec}$)

$simDPF_{dec}$ is more fine grained than $simDPF_{sus}$, and considers that when a unit is destroyed, the DPF that a player can do is reduced. Thus, instead of computing how much time it will take to destroy the other army, we only compute how much time it will take to kill one unit, selected by the *target selection* function. Then, the unit that was killed is subtracted from the state, and we recompute the time to kill the survivors and which will be the next targeted unit. This is an iterative way to remove units and keep updating the current DPF of the armies. The process is detailed in Algorithm 1, where first the model determines which is the next unit that each player will attack using a *target selection* policy (lines 4-7); after that we compute the expected time to kill the selected unit using TIMEKILLUNIT(u, G, DPF); and the target that should be killed first is eliminated from the state (only one unit of the group), and the HP of the survivors is updated (lines 18-25). We keep doing this until one army is completely annihilated or we cannot kill more units.

Notice that $simDPF_{dec}$ has the same three input parameters as $simDPF_{sus}$. Let us now focus on how can those parameters be acquired for a given RTS game.

Algorithm 1 Combat Simulator using decreased DPF over time.

1: **function** SIMDPFDEC$(G, DPF, targetSelection)$
2: $E \leftarrow \{g \in G | g.player = p_1\}$ ▷ enemy units
3: $F \leftarrow \{g \in G | g.player = p_2\}$ ▷ friendly units
4: SORT$(E, targetSelection)$
5: SORT$(F, targetSelection)$
6: $e \leftarrow$ POP(E) ▷ pop first element
7: $f \leftarrow$ POP(F)
8: **while true do**
9: $t_e \leftarrow$ TIMEKILLUNIT(e, F, DPF)
10: $t_f \leftarrow$ TIMEKILLUNIT(f, E, DPF)
11: **while** $t_e = \infty$ **and** $E \neq \emptyset$ **do**
12: $e \leftarrow$ POP(E)
13: $t_e \leftarrow$ TIMEKILLUNIT(e, F, DPF)
14: **while** $t_f = \infty$ **and** $F \neq \emptyset$ **do**
15: $f \leftarrow$ POP(F)
16: $t_f \leftarrow$ TIMEKILLUNIT(f, E, DPF)
17: **if** $t_e = \infty$ **and** $t_f = \infty$ **then break** ▷ to avoid a deadlock
18: **if** $t_e < t_f$ **then**
19: **if** $E = \emptyset$ **then break** ▷ last unit killed
20: $e \leftarrow$ POP(E)
21: $f.HP \leftarrow f.HP -$ DPF$(E) \times t_e$
22: **else**
23: **if** $F = \emptyset$ **then break** ▷ last unit killed
24: $f \leftarrow$ POP(F)
25: $e.HP \leftarrow e.HP -$ DPF$(F) \times t_f$
26: **return** $E \cup F$

Model Parameters

As we can observe our two proposed models have three main parameters.

- **Unit hit points**. The maximum hit points of each unit is something we know beforehand and invariant during the game. Therefore there is no need to learn this parameter.
- **Unit DPF**. There is a theoretical (maximum) DPF that a unit can deal, but this value is highly affected by the time between shots, which heavily depends on the maneuverability and properties of units as compared to the targets, and on the skill of the player to control the units. Therefore we encode this as a $n \times n$ DPF matrix, where $DPF(i,j)$ represents the DPF that a unit of type i usually deals to a unit of type j. We call this the *effective DFP*.
- **Target selection**. When two groups containing units of different types face each other, determining which types of units to attack first is key. In theory, determining the optimal attack order is an EXPTIME problem (Furtak and Buro 2010). Existing models of StarCraft such as SparCraft model this by having a portfolio of handcrafted heuristic strategies (such us attack closest or attack unit with highest DPF / HP) which the user can configure for each simulation. We propose to train the target selection policy from data. In order to obtain a forward model that predicts the expected outcome of a combat given usual player behavior.

Thus, the parameters that we are tying to learn are the DPF matrix and the *target selection* policy. To do so, we collected a dataset which we describe below. We argue that these two parameters are enough to capture the dynamics of a range of RTS games to a sufficient degree for resulting in accurate forward models.

Dataset

The parameters required by the models proposed above can automatically be acquired from data. In particular, the dataset can be generated directly by processing replays, a.k.a. game log files. StarCraft replays of professional player games are widely available, and several authors have compiled collections of such replays in previous work (Weber and Mateas 2009; Synnaeve and Bessière 2012).

Since StarCraft replays only contain the mouse click events of the players (this is the minimum amount of information needed to reproduce the game in StarCraft), we don't know the full game state at a given point (no information about the location of the units or their health). This small amount of information in the replays is enough to learn many aspects of RTS game play, such as build orders (Hsieh and Sun 2008) or the expected rate of unit production (Stanescu and Certicky 2014). However, it is not enough to train the parameters we require in our models.

Thus, we need to run the replay in StarCraft and record the required information using BWAPI[1]. This is not a trivial task since if we record the game state at each frame we will have too much information (some consecutive recorded game state can be the same) and we will need a lot of space to store all this information. Some researchers proposed to capture different information at different resolutions to have a good trade-off of information resolution. For example, Synnaeve and Bessière (2012) proposed recording information at three different levels of abstraction:

- General data. Records all BWAPI events (like unit creation, destruction, discovery, ...). Economical situation every 25 frames and attack information every frame. It uses a heuristic to detect when an attack is happening.
- Order data. Records all orders to units. It is the same information you will get parsing the replay file outside BWAPI.
- Location data. Records the position of each unit every 100 frames.

On the other hand, Robertson and Watson (2014) proposed a uniformed information gathering, recording all the information every 24 frames or every 7 frames during attacks to have a better resolution than the previous work.

In our case we only need the combat information, so we decided to use the analyzer from Synnaeve and Bessière, but with an updated heuristic for detecting combats (described below), since theirs was not enough for our purposes. Therefore we implemented our own method to detect combats[2]. Since, we are interested in capturing the information of *when* and *where* a combat started, *what* units were destroyed, and the *initial* and *final* army composition. We define a combat as a tuple $C = \langle t_s, t_f, R, U_0, U_1, A_s^0, A_s^1, A_f^0, A_f^1, K\rangle$, where:

- t_s is the frame when the combat started and t_f the frame when it finished,
- R is the reason why the combat finished. The options are:
  - **Army destroyed**. One of the two armies was totally destroyed during the combat.
  - **Peace**. None of the units were attacking for the last x frames. In our experiments $x = 144$, which is 6 seconds of game play.
  - **Reinforcement**. New units participating in the battle. This happens when units, that were far from the battle when it started, begin to participate in the combat. Notice that if a new unit arrives but never participates (it does not attack another unit) we do not consider it as a reinforcement.
  - **Game end**. Since a game can end from one of the players surrendering, at the end of the game we close all the open combats.
- U_i is a list of upgrades and technologies researched by player i at the moment of the combat. $U = \{u_1, \ldots, u_n\}$ where u_j is an integer denoting the level of the upgrade type j.
- A_w^p where $w \in \{s, f\}$ is the high-level state of the army of player p at the start of the combat (A_s^p) and at the end (A_f^p). An army is a list of tuples $A = \langle id, t, p, hp, s, e\rangle$ where id is an identifier, t is the unit type, $p = (x, y)$ is a position, hp the hitpoints, s the shield, and e the energy.
- $K = \{(t_1, id_1), \ldots (t_n, id_n)\}$ is a list of game time and identifier of each unit killed during the combat.

In Synnaeve and Bessière's work the beginning of a combat is marked when a unit is killed. For us this is too late since one unit is already dead and several units can be injured during the process. Instead, we start tracking a new combat if a **military unit** is **aggressive** or **exposed** and not already in a combat. Let us define some terms in our context. A **military unit** is a unit that can attack or cast spells, detect cloaked units, or a transporter. We call a unit is **aggressive** when it has the order to attack or is inside a transport. A unit is **exposed** if it has an *aggressive* enemy unit in attack range.

Any unit u belonging to player p_0 meeting these conditions at a time t_0 will trigger a new combat. Let us define $inRange(u)$ to be the set of units in attack range of a unit u. Now, let $A = \cup_{u' \in inRange(u)} inRange(u')$. A_s^0 is the subset of units of A belonging to player p_0 and A_s^1 is the subset of units of A belonging to the other player (p_1). Figure 2 shows a representation of a combat triggered a unit u (the black filled square) and the units in the combat (the filled squares). Notice that a *military unit* is considered to take part in a combat even if at the end it does not participate.

By processing a collection of replays and storing each combat found, a dataset is built to train the parameters of our models.

[1] https://github.com/bwapi/bwapi

[2] Source code of our replay analyzer can be found at https://bitbucket.org/auriarte/bwrepdump

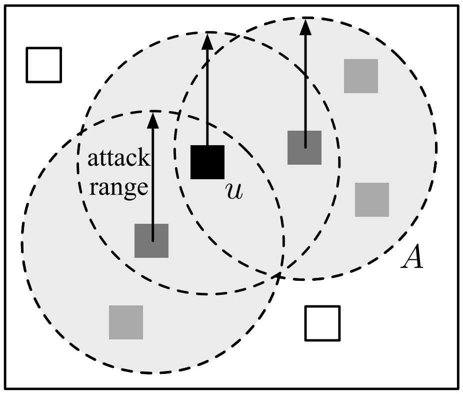

Figure 2: Black filled square triggers a new combat. Only filled squares are added to the combat tracking.

Learning the DPF Matrix

The main idea is to estimate a DPF matrix of size $n \times n$, where n is the number of different unit types (in StarCraft $n = 163$). For each combat in our dataset, we perform the following steps:

1. First, for each player we count how many units can attack ground units ($size_{ground}$), air units ($size_{air}$) or both ($size_{both}$).
2. Then for each kill in $(t_n, id_n) \in K$ we compute the total damage done to the unit killed (id_n) as: $damage = id_n.HP + id_n.shield$ where *HP* and *shield* are the hit points and shield of the unit at the start of the combat. This damage is split between all the units that could have attacked id_n. For instance, if id_n is an air unit the damage is split as:
$$damageSplit = \frac{damage}{size_{air} + size_{both}}$$
notice that the damage is split even if some of the units did not attack id_n, since in our dataset we do not have information of which units actually did attack id_n. After that, for each unit id_{attack} that could attack id_n, we update two global counters:
$$damageToType(id_{attack}, id_n) += damageSplit$$
$$timeAttackingType(id_{attack}, id_n) += t_s - t_n$$
3. After parsing all the combats, we compute the DPF that a unit of type i usually deals to a unit of type j as:
$$DPF(i,j) = \frac{damageToType(i,j)}{timeAttackingType(i,j)}$$

Even if this process has some inaccuracies (we might be assigning damage to units who did not participate, not considering HP or shield regeneration or friendly damage), our hypothesis is that with enough training data, these values should converge to realistic estimates of the effective DPF.

Learning Target Selection

To learn the target preference we use the Borda count method to give points towards a unit type each time we make chose. So, the idea is to iterate over all the combats and each time we kill for the first time a type of unit we give that type $n - i$ points where n is the number of different unit types in the group and i the order the units were killed. For example, if we are fighting against marines, tanks and workers ($n = 3$) and we killed first the tank then the marines and last the workers, the scores will be: $3-1 = 2$ points for the tank, $3 - 2 = 1$ point for the marines and $3 - 3 = 0$ points for the workers. After analyze all the combats we compute the average Borda count and this is the score we use to sort our targets in order of preference.

Experimental Evaluation

In order to evaluate the performance of each simulator ($simDPF_{sus}$, $simDPF_{dec}$ and SparCraft), we used different configurations (with hardcoded parameters and with trained parameters). We compare the generated predictions with the real outcome of the combats collected in our dataset. The following subsections present our experimental setup and the results of our experiments.

Experimental Setup

We extracted the combats from 49 Terran vs Terran and 50 Terran vs Protoss replay games. This resulted in:

- 1,986 combats ended with one *army destroyed*,
- 32,975 combats ended by *reinforcement*,
- 19,648 combats ended in *peace*,
- 196 combats ended by *game end*.

We are only interested on the combats ended by one army destroyed, since those are the scenarios that are most informative. We also removed combats with Vulture's mines (to avoid problems with friendly damage) and with transports. This resulted in a dataset with 1,565 combats.

We compared different configurations of our $simDPF_{sus}$ and $simDPF_{dec}$ combat forward models to compare the results:

- DPF: we experimented with two DPF matrices. DPF_{data}: calculated directly from the weapon damage and cooldown of each unit in StarCraft. DPF_{learn} learned from traces, as described above.
- Target selection: we experimented with three target selection policies. TS_{random}: randomly selecting the next unit (made deterministic for experimentation by using always the same seed). TS_{ks}: choosing always the unit with the highest *kill score* (this is an internal StarCraft score based on the resources needed to produce the unit). TS_{learn}: automatically learned from traces, as described above.

We evaluate our approach using a 10-fold cross-validation. After training is completed we simulate each test combat with our models using different configurations. Once we have a combat prediction from each simulator, we compare it against the real outcome of the combat from the

Table 2: Average Jaccard Index of the combat models with different configurations over 1,565 combats.

	DPF_{data} TS_{random}	DPF_{data} TS_{ks}	DPF_{learn} TS_{learn}
$simDPF_{sus}$	0.861	0.861	0.848
$simDPF_{dec}$	0.899	0.905	0.888

dataset. This comparison uses a modified version of the Jaccard index, as described below. The Jaccard index is a well known similarity measure between sets (the size of their intersection divided by the size of their union).

In our experiments we have an initial game state (A), the outcome of the combat from the dataset (B), and the result of our simulator (B'). As defined above, our high-level abstraction represents game states as a set of unit groups $A = \{a_1, \ldots, a_n\}$, where each group has a player, a size and a unit type. In our similarity computation, we want to give more importance if a unit from a small group is missing than another from a bigger group (two states are more different if the only Siege tank was destroyed, than if only one out of 10 marines was destroyed). Thus, we compute a weight for each unit group a_k in the initial state A, as:

$$w_k = \frac{1}{a_k.size + 1}$$

The similarity between the prediction of our forward model (B'), and the actual outcome of the combat in the dataset (B) is defined as:

$$J(A, B', B) = \frac{\sum_{k=1}^{n} (min(b_k.size, b'_k.size) \times w_k)}{\sum_{k=1}^{n} (max(b_k.size, b'_k.size) \times w_k)}$$

As mentioned before, we use SparCraft to have another baseline to compare our proposed combat models. SparCraft comes with several scripted behaviors. In our experiments we use the following: *Attack-Closest (AC)*, *Attack-Weakest (AW)*, *Attack-Value (AV)*, *Kiter-Closest (KC)*, *Kiter-Value (KV)*, and *No-OverKill-Attack-Value (NOK-AV)*.

Results

Table 2 shows the average similarity (computed using the Jaccard index described above) of the predictions generated by our combat models with respect to the actual outcome of the combats. The first thing we see is that the predictions made by all our models are very similar to the ground truth: Jaccard indexes higher than 0.86, which are quite high similarity values. There is an important and statistically significant difference (p-value of 3.5×10^{-10}) between $simDPF_{sus}$ and $simDPF_{dec}$. While the different configurations of $simDPF_{dec}$ achieve similar results. Using a predefined DPF matrix and a kill score-based target selection achieves the best results. However, we see that in domains where DPF information or target selection criteria (such as kill score) is not available, we could learn them from data, and achieve very similar performance (as shown on the right-most column).

Table 3: Average Jaccard Index and time of different combat models over 328 combats.

Combat Model	Avg. Jaccard	Time (sec)
$simDPF_{sus}$ $(DPF_{learn}, TS_{learn})$	0.874	0.03321
$simDPF_{dec}$ $(DPF_{learn}, TS_{learn})$	0.885	0.03919
SparCraft (AC)	0.891	1.68190
SparCraft (AW)	0.862	1.36493
SparCraft (AV)	0.874	1.47153
SparCraft (NOK-AV)	0.875	1.35868
SparCraft (KC)	0.846	6.87307
SparCraft (KV)	0.850	6.95467

To compare our model to previous work we also run our experiments in SparCraft. Since SparCraft does not support all units we need to filter our initial set of 1,565 combats removing those that are incompatible with SparCraft. This results in a dataset of 328 combats where the results can be shown at Table 3. This shows that our combat simulator $simDPF_{dec}$ has similar performance to the SparCraft configuration (AC) that achieves better results, but it is 43 times faster. This can be a critical feature if we are planning to execute thousands of simulations like it will happen if we use it as a forward model in a MCTS algorithm. Also, notice that $simDPF_{sus}$ performs better in this dataset than in the complete dataset, since the 328 combats used for this second experiments are simpler.

Conclusions

The long-term goal of the work presented in this paper is to design game-playing systems that can automatically acquire forward models for new domains, thus making them both more general and also applicable to domains where such forward models are not available. Specifically, this paper presented two alternative forward models for StarCraft combats, and a method to train the parameters of these models from replay data. We have seen that in domains where the parameters of the models (damage per frame, target selection) are available from the game definition, those can be used directly. But in domains where that information is not available, it can be estimated from replay data.

Our results show that the models are as accurate as hand-crafted models such as SparCraft for the task of combat outcome prediction, but much faster. This makes our proposed models suitable for MCTS approaches that need to perform a large number of simulations.

As part of our future work we would try to improve our combat simulator. For example, we could incorporate the ability to spread the damage done through different group types instead of all our groups attacking the same group type. Additionally, we would like to experiment using the simulator with different configurations inside a MCTS tactical planner in the actual StarCraft game, to evaluate the performance that can be achieved using our trained forward model, instead of using hard-coded models.

References

Balla, R.-K., and Fern, A. 2009. UCT for tactical assault planning in real-time strategy games. In *International Joint Conference of Artificial Intelligence (IJCAI 2009)*, 40–45.

Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A survey of monte carlo tree search methods. *Transactions on Computational Intelligence and AI in Games (TCIAIG)* 4(1):1–43.

Buro, M. 2003. Real-time strategy games: a new ai research challenge. In *International Joint Conference of Artificial Intelligence (IJCAI 2003)*, 1534–1535. Morgan Kaufmann Publishers Inc.

Churchill, D., and Buro, M. 2013. Portfolio greedy search and simulation for large-scale combat in StarCraft. In *Symposium on Computational Intelligence and Games (CIG 2013)*. IEEE.

Churchill, D.; Saffidine, A.; and Buro, M. 2012. Fast heuristic search for RTS game combat scenarios. In *Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2012)*. AAAI Press.

Furtak, T., and Buro, M. 2010. On the complexity of two-player attrition games played on graphs. In Youngblood, G. M., and Bulitko, V., eds., *Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2010)*. AAAI Press.

Hsieh, J.-L., and Sun, C.-T. 2008. 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*, 3106–3111. IEEE.

Jaidee, U., and Muñoz-Avila, H. 2012. CLASSQ-L: A q-learning algorithm for adversarial real-time strategy games. In *Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2012)*. AAAI Press.

Justesen, N.; Tillman, B.; Togelius, J.; and Risi, S. 2014. Script- and cluster-based UCT for StarCraft. In *Symposium on Computational Intelligence and Games (CIG 2014)*. IEEE.

Ontañón, S.; Synnaeve, G.; Uriarte, A.; Richoux, F.; Churchill, D.; and Preuss, M. 2013. A survey of real-time strategy game Ai research and competition in starcraft. *Transactions on Computational Intelligence and AI in Games (TCIAIG)* 5:1–19.

Perkins, L. 2010. Terrain analysis in real-time strategy games: An integrated approach to choke point detection and region decomposition. In *Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2010)*. AAAI Press.

Robertson, G., and Watson, I. 2014. An improved dataset and extraction process for starcraft ai. In *The Twenty-Seventh International Flairs Conference*.

Soemers, D. 2014. Tactical planning using MCTS in the game of StarCraft. Master's thesis, Department of Knowledge Engineering, Maastricht University.

Stanescu, M., and Certicky, M. 2014. Predicting opponent's production in real-time strategy games with answer set programming. *Transactions on Computational Intelligence and AI in Games (TCIAIG)*.

Stanescu, M.; Hernandez, S. P.; Erickson, G.; Greiner, R.; and Buro, M. 2013. Predicting army combat outcomes in starcraft. In *Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2013)*. AAAI Press.

Synnaeve, G., and Bessière, P. 2012. A dataset for StarCraft AI & an example of armies clustering. In *Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2012)*. AAAI Press.

Uriarte, A., and Ontañón, S. 2014. Game-tree search over high-level game states in RTS games. In *Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2014)*. AAAI Press.

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