Artificial Intelligence in Adversarial Games: Papers from the AIIDE Workshop
AAAI Technical Report WS-16-21

Experiments on Learning Action Probability Models from Replay Data in RTS Games

Santiago Ontañón
Computer Science Department
Drexel University
santi@cs.drexel.edu

Abstract

Recent work has shown that incorporating action probability models (models that given a game state can predict the probability with which an expert will play each move) into MCTS can lead to significant performance improvements in a variety of adversarial games, including RTS games. This paper presents a collection of experiments aimed at understanding the relation between the amount of training data, the predictive performance of the action models, the effect of these models in the branching factor of the game and the resulting performance gains in MCTS. Experiments are carried out in the context of the μRTS simulator, showing that more accurate predictive models do not necessarily result in better MCTS performance.

Introduction

Informed Monte Carlo Tree Search algorithms are based on incorporating *action probability* models into Monte Carlo Tree Search (MCTS) in order to bias exploration of the tree. Recent work has shown that these approaches can lead to significant performance improvements in a variety of games. For example, Silver et al. (Silver et al. 2016) showed improvements in game-play performance in the game of Go, defeating one of the world's top players. In our recent work (Ontañón 2016; Uriarte and Ontañón 2016), we have shown that significant gains can also be achieved in real-time strategy games such as StarCraft and μRTS with models learned using Bayesian Networks.

The goal of this paper is to gain a better understanding of the relation between the action probability models and the game-play performance achieved by MCTS in real-time strategy (RTS) games. Specifically, this paper is a follow-up to our work on using Bayesian models to learn action probability models for RTS games (Ontañón 2016). While in our previous work, we showed the feasibility of using Bayesian models to learn action probability models that make MCTS perform significantly better, in this paper we present experiments varying the conditions under which the Bayesian models are learned, to understand the different aspects that affect the performance of *informed MCTS*. All the experiments are carried out in the context of the μRTS simulator.

Results show that, as expected, models trained from better experts are stronger, more training data results in stronger models, and that using richer feature sets results in more accurate models. However, surprisingly, our results indicate that more accurate models do not necessarily result in stronger MCTS performance. Our hypothesis is that when an action probability model concentrates the probability mass of the prediction in only very few actions, this narrows down the search space of MCTS too much, resulting in weaker gameplay. We also hypothesize that ϵ-greedy (which is the basis of the bandit policy used in our experiments) is particularly affected by this, and perhaps other bandit policies would be less affected.

The remainder of this paper is organized as follows. We first provide some background on RTS games, followed by a brief description of the Bayesian action probability model used in our experiments. After that, we present a series of five experiments to understand the performance of these models and of informed MCTS in different circumstances.

Background

Real-time Strategy (RTS) games are complex adversarial domains, typically simulating battles between a large number of military units, that pose a significant challenge to both human and artificial intelligence (Buro 2003). Designing AI techniques for RTS games is challenging because (1) they have *huge decision spaces*: the branching factor of a typical RTS game, StarCraft, has been estimated to be on the order of 10^{50} or higher (Ontanón et al. 2013); and (2) they are *real-time*, which means that these games typically execute at 10 to 50 decision cycles per second, leaving players with just a fraction of a second to decide the next action, players can issue actions simultaneously, and actions are durative.

The reason for which the branching factor in some RTS games is so large is that players control many units, and players can issue multiple actions at the same time (one per unit). We will refer to those actions as *unit-actions*. A *player-action* is the set of unit-actions that one player issues simultaneously in a given game cycle. Thus players issue only one *player-action* at any given time (which will consist of zero or more unit-actions). To illustrate this, consider the situation from the μRTS game shown in Figure 1. Two players, *max* (shown in blue) and *min* (shown in red) control 9 units each. Consider the bottom-most circular unit

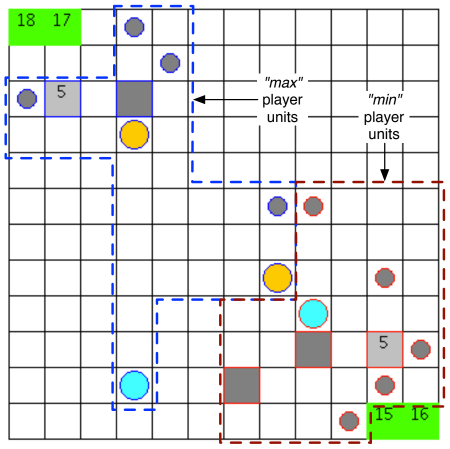

Figure 1: A screenshot of μRTS. Square units are "bases" (light grey, that can produce workers), "barracks" (dark grey, that can produce military units), and "resources mines" (green, from where workers can extract resources to produce more units), the circular units are "workers" (small, dark grey) and military units (large, yellow or light blue).

in Figure 1 (a worker). This unit can execute 8 actions: stand still, move left or up, harvest the resource mine to the right, or build a barracks or a base in any of the two adjacent cells. In total, player *max* can issue 1,008,288 different player-actions, and player *min* can issue 1,680,550 different player-actions. Thus, even in relatively simple scenarios, the branching factor is very large.

We will define an RTS game as a tuple $G = (P, S, A, L_u, L_p, T, W, s_{init})$, where:

- $P = \{max, min\}$ is the set of players.
- S is the set of possible states. We will write $units(p, s)$ as the set of units that belong to player p in state s.
- A is the finite set of unit-actions that units can execute.
- $L_u(u, a, s) \rightarrow \{true, false\}$, is a function that returns whether unit u can execute unit-action a in state s. For simplicity, we will write $L_u(u, s) = \{a \in A | L_u(u, a, s) = true\}$, and $ready(p, s) = \{u \in units(p, s) | L_u(u, s) \neq \emptyset\}$.
- $L_p(p, \alpha, s) \rightarrow \{true, false\}$, is a function that returns whether player p can execute player-action α in state s. Given the set of ready units $ready(p, s) = \{u_1, ..., u_n\}$, a player-action α is defined as $\alpha = \{(u_1, a_1), ..., (u_n, a_n)\}$, such that $L_u(u_i, a_i, p) = true$ for $1 \leq i \leq n$. Thus, the $ready$ function determines the set of units that can execute unit-actions, L_u determines which actions can each of those units execute, which determines the set of possible player-actions, and L_p determines which of those possible player-actions is legal.
- $T(s_t, \alpha_{min}, \alpha_{max}) \rightarrow S$ is the deterministic transition function, that given a state $s_t \in S$ at time t, and the player-actions of each player (α_{min} and α_{max}), returns the state that will be reached at time $t + 1$ (i.e., T is the *forward model* of the game).
- $W : S \rightarrow \{maxwins, minwins, draw, ongoing\}$ is a function that determines the winner of the game, if the game is still ongoing, or if it is a draw.
- $s_{init} \in S$ is the initial state.

Action Probability Models in MCTS

An *action probability model* captures the conditional probability distribution $P(A|S)$ of the action A a player would perform given a game state S. *AlphaGO*'s *policy network* is an example of such action probability model. In the context of RTS games, we could define action probability models in different ways. For example, we could define *unit-action probability models* (that capture the probability distribution of the actions individual units perform) (Ontañón 2016), *squad-action probability models* (that capture the distribution of actions that groups of units perform) (Uriarte and Ontañón 2016), or *player-action probability models* (that capture the distribution of player actions). In this paper we will focus on unit-action probability models.

This section introduces a model validated in our previous work to be useful both for unit-action models and for squad-action models (Ontañón 2016; Uriarte and Ontañón 2016). This model is based on the idea of the Naive Bayes classifier (Rish 2001), and thus, has negligible training time.

Action-Type Interdependence Model (AIM)

Given a game state s, a player p, and a unit $u \in ready(p, s)$, we would like to model the probability $P(a|s, u)$ with which an expert would select each of the actions $a \in L_u(u, s)$.

In order to model such probability distribution, we assume that the game state s (from the perspective of u) is represented by means of a feature vector $\mathbf{x}(u, s) = \{x_1(u, s), ..., x_n(u, s)\}$ of length n, and that the distribution is estimated from a training set $I = \{(s, u, a), ...\}$, where each training instance has a game state s, a unit u, and the unit-action a that was chosen by an expert. In the experiments below, we report results with different feature sets.

While the Naive Bayes classifier often works very well for classification purposes, it is well known that the probability distribution it estimates is not well "calibrated" (Bennett 2000), i.e., values tend to be very extremely close to either 0 or 1. To correct for this, we introduce a calibration parameter $\kappa > 0$ into the model formulation.

In order to reduce the number of parameters to estimate from data, we assume the existence of a function $type(a)$, which assigns a *type* to an action a from a predefined set of action types (e.g., move, attack, etc.). So, even if actions such as "move up" and "move down" are different actions, they both have the same type, "move". Let us define $legaltypes_u(u, s) = \{type(a) | a \in L_u(u, s)\}$ as the set of action types that unit u can perform in state s. The **AIM** model is defined as follows:

$$P(a|u, s) = \frac{1}{Z} \left(P(a)\ L(type(a), T)\ F(a, u, s)\right)^c$$

where Z is just a normalization constant to make all the probabilities add up to 1, $c = \frac{1}{1+\kappa(n+|T|)}$, and κ is a calibration parameter, whose effect is to make the probability values less extreme. $T = \mathit{legaltypes}_u(u, s)$, and $F(a, s)$ is the product of the factors contributed by the features in $\mathbf{x}(s)$:

$$F(a, u, s) = \prod_{i=1\dots n} P(x_i(u, s)|a)$$

Finally, $L(\mathit{type}(a), T)$ captures the probability that a certain unit-action type is legal, given the type of the unit-action that was selected:

$$L(t, T) = \prod_{t' \in T} P(t' \text{ is legal}|t \text{ was selected})$$

Here, $P(t' \text{ is legal}|t \text{ was selected})$ is the probability that an action of type t' was legal in a game state where an action of type t was selected. $P(t' \text{ is legal}|t \text{ was selected})$, $P(x_i(u, s)|a)$ and $P(a)$ are all estimated from the training set[1], and κ is determined via simple grid search using the training set, testing values between 0.0 to 1.0 at intervals of 0.05, and keeping the value that maximizes the likelihood of the training data given the model.

Moreover, as reported in our previous work (Ontañón 2016), learning a different model for each different unit type in the game (*workers*, *bases*, *barracks*, etc. in μRTS), resulted in better estimation of the probabilities. For the experiments reported in this paper, we train a model per unit type using the subset of the training data referring to such unit type. If this subset is empty, then we use the whole training set (i.e., if we have no training data to model the way a specific unit is controlled, we just train a model with the whole training set for such unit, hoping it will reflect what the expert would have done).

Extending Unit-Action to Player-Action Distributions

When a player-action for player p needs to be generated in a game state s according to a unit-action probability model that generates a probability distribution P, we use the following procedure:

1. Push all the units $\mathit{ready}(p, s)$, to a queue Q in a random order. Initialize an empty player-action $\alpha = \emptyset$.
2. If Q is empty, return α.
3. Otherwise, remove the first unit u from Q. Let $l = \{a \in L_u(u, s)|L_p(p, \alpha \cup (u, a), s)\}$, i.e., the set of legal unit-actions for u that when added to the player-action α still keep α being legal.
4. If $l = \emptyset$, restart the process from 1.
5. Otherwise, sample one action a from l according to P, add it to α as: $\alpha = \alpha \cup (u, a)$, and go back to 2.

The previous process samples a player-action using the unit-action distribution P, while respecting unit-action legality (L_u) and player-action legality (L_p).

[1]All probability estimations from the training set were estimated using Laplace estimation. For example, when estimating $P(a)$, we add 1 to the numerator, and $|A|$ to the denominator, resulting in $P(a) = \frac{\text{number of times } a \text{ is selected}+1}{\text{size of the training set}+|A|}$

Informed Monte Carlo Tree Search

We incorporated the models described above into Monte Carlo Tree Search (MCTS), a family of planning algorithms based on sampling the decision space rather than exploring it systematically (Browne et al. 2012). MCTS employs two different *policies* to guide the search: (1) a *tree policy* determines which nodes in the tree to explore (i.e., given a node in the tree, which of its children to consider next), and, (2) each time a new node is added to the tree, a simulation (a *playout* or *rollout*) of how the game would unfold from that state until the end of the game (or until a predefined maximum playout length) is executed by using a *default policy* to generate actions for both players.

Thus, the action probability models learned above can be used in MCTS in two ways: to define *tree policies* or *default policies*. While an action probability model can be used directly as a default policy, in order to be used as a *tree policy*, it needs to be incorporated into a multi-armed bandit policy.

Informed ϵ-Greedy sampling. As any MAB policy, *informed ϵ-greedy sampling* will be called many iterations in a row. At each iteration t, an action $a_t \in A$ is selected, and a reward r_t is observed.

Given $0 \geq \epsilon \geq 1$, a finite set of actions A to choose from, and a probability distribution P, where $P(a)$ is the a priori probability that a is the action an expert would choose, *informed ϵ-greedy sampling* works as follows:

- Let us call $\overline{r}_t(a)$ to the current estimation (at iteration t) of the expected reward of a (i.e., the average of all the rewards obtained in the subset of iterations from 0 to $t-1$ where a was selected). By convention, when an action has not been selected before t we will have $\overline{r}_t(a) = 0$.
- At each iteration t, action a_t is chosen as follows:
  - With probability ϵ, choose a_t according to the probability distribution P.
  - With probability $1 - \epsilon$, choose the best action so far: $a_t = argmax_{a \in A}\overline{r}_t(a)$ (ties resolved randomly).

In order to test the proposed models in the context of RTS games, we use a modification of the NaiveMCTS algorithm (Ontanón 2013). NaiveMCTS is a MCTS algorithm designed to handle RTS games: it supports durative and simultaneous actions and uses a bandit strategy called *naive sampling*, which handles the combinatorial number of actions in RTS games. Naive sampling internally uses a collection of ϵ-greedy sampling strategies. For the experiments in this paper, the modification of NaiveMCTS consisted in replacing these ϵ-greedy sampling strategies by informed ϵ-greedy. We call the resulting algorithm *INMCTS*.

Let us now present a series of experiments to evaluate several key aspects concerning the application of action probability models to MCTS algorithms.

Training Data

We generated training data following the same procedure as (Ontañón 2016): we selected six bots built-in into μRTS: (*LSI* (Shleyfman, Komenda, and Domshlak 2014), *NaiveMCTS* (Ontanón 2013)), *WorkerRush*, *LightRush*, *HeavyRush*, and *RangedRush*), and played a round-robin tournament (all 36 combinations of each of the 6 bots playing as player 1 and as player 2) in 8 different maps[2], resulting in a total of $288 = 36 \times 8$ games. The configuration used for NaiveMCTS and LSI was the default one as implemented in μRTS, where playouts are limited to be at most 100 game frames long, after which an evaluation function is applied.

We repeated this round-robin tournament four times giving *NaiveMCTS* and *LSI* a computation budget of 500, 1000, 2000, 5000, and 10000 playouts per game frame respectively, resulting in five sets of game logs used as training data for the experiments reported in the rest of this paper.

We experimented with three different ways of calculating the feature vector $\mathbf{x}(u, s)$ used to represent each game state:

- *fs1*: no features at all, $\mathbf{x}(u, s) = \emptyset$.
- *fs2*: $\mathbf{x}(u, s)$ composed of eight features: the number of resources available to the player, the cardinal direction (north, east, south, west) toward where most friendly units are, the cardinal direction toward where most enemy units are, whether we have a barracks or not, and four features indicating the type of the unit in the cell two positions north, east, south or west.
- *fs3*: $\mathbf{x}(u, s)$ composed of 46 features, which include the eight above, but also: hit points, attack range and resources of the unit, the content of all the cells in a radius of 3 cells, plus a few additional combined features[3].

Unless otherwise specified, by default we use set *fs2*.

Experiment 1: Source of the Training Data

In this first experiment, we evaluate the performance that can be expected from the proposed model given different sets of training data (generated by different bots). Table 1 shows:

- *Exp. LL*: the accuracy of the model in predicting the actions of the bot (evaluated using a 10 fold cross validation), and measured as the expected log-likelihood of the actual action the bot performed. A value of 0 would mean perfect predictions, and lower values represent worse predictions. As the table shows, the *WorkerRush*, *LightRush*, *HeavyRush*, and *RangedRush* which implement hard-coded strategies can be predicted more accurately than LSI and NaiveMCTS. Moreover, LSI and NaiveMCTS become more predictable the larger their computational budget, which is also expected.
- *Gameplay Strength (Original)*: the gameplay strength of the original bot (average score where winning counts as 1 point, and ties count 0.5 points). This is evaluated by making each AI play against each of the six AIs used for generating the training data, plus the two random bots in all eight maps (repeating each match 10 times). As we can see, LSI and NaiveMCTS achieve the highest performance, specially when given a larger computation budget, but performance seems to plateau when given a budget of 5000 or more playouts. For comparison, we include the gameplay strength of the two stochastic bots that come built-in into μRTS (*Random* and *RandomBiased*).
- *Gameplay Strength (***AIM***)*: the gameplay strength when we take the **AIM** model and use it to directly play the game (sampling actions stochastically following the model's probability distribution, **without** using MCTS). We see that the model learned from *WorkerRush* achieves the highest performance (probably because this is a very aggressive and deterministic bot, which not too hard to predict). Training from LSI and NaiveMCTS, we see that the model performs better when training from bots with larger computation budgets. This is probably since the bots converge to a more stable strategy, easier to predict.

[2]Specifically, we used the maps *OneBaseWorker8x8*, *TwoBasesWorkers8x8*, *ThreeBasesWorkers8x8*, *FourBasesWorkers8x8*, *OneBaseWorker12x12*, *TwoBasesWorkers12x12*, *ThreeBasesWorkers12x12*, and *FourBasesWorkers12x12* included with μRTS.

[3]Source code with the specific features available at: http://www.cs.drexel.edu/~santi/AIIDE-RTSWS-2016-microRTS-source-code.zip

Table 1: Accuracy and gameplay strength of the **AIM** models compared to the bots on which they were trained.

		Gameplay Strength	
Bot	Exp. LL	Original	**AIM**
Random	-	0.077	-
RandomBiased	-	0.197	-
WorkerRush	-0.892	0.764	0.518
LightRush	-0.441	0.602	0.211
HeavyRush	-0.345	0.440	0.161
RangedRush	-0.318	0.446	0.139
LSI (500)	-1.229	0.788	0.187
LSI (1000)	-1.211	0.839	0.218
LSI (2000)	-1.193	0.886	0.213
LSI (5000)	-1.175	0.921	0.247
LSI (10000)	-1.146	0.921	0.284
NaiveMCTS (500)	-1.206	0.837	0.197
NaiveMCTS (1000)	-1.193	0.879	0.205
NaiveMCTS (2000)	-1.181	0.902	0.227
NaiveMCTS (5000)	-1.169	0.938	0.247
NaiveMCTS (10000)	-1.140	0.935	0.292

Experiment 2: Feature Set

Table 2 shows the accuracy of the **AIM** models, as well as their gameplay strength when using different feature sets. As expected, when using more features, the resulting models are more accurate (lower expected loglikelihood), and also tend to achieve higher gameplay strength. Moreover, as we will show in Experiment 5, when incorporating these models into MCTS, the resulting performance of MCTS does not increase accordingly, which was unexpected to us.

Experiment 3: Amount of Training Data

This section presents an experiment concerning the amount of training data, in order to determine whether we generated enough training data for our models. Table 3 shows that performance when training from a small amount of data (10% of the training set) is already almost identical to that achieved with 100% of the data (although performance does improve slightly with more training data). Results indicate

Table 2: Accuracy and gameplay strength of **AIM** models trained with different feature sets.

Expected Loglikelihood			
Bot	**AIM** (fs1)	**AIM** (fs2)	**AIM** (fs3)
WorkerRush	-1.146	-0.892	-0.780
LSI (500)	-1.245	-1.229	-1.217
LSI (10000)	-1.198	-1.146	-1.115
NaiveMCTS (500)	-1.225	-1.206	-1.195
NaiveMCTS (10000)	-1.173	-1.140	-1.113
Gameplay Strength			
Bot	**AIM** (fs1)	**AIM** (fs2)	**AIM** (fs3)
WorkerRush	0.288	0.518	0.530
LSI (500)	0.110	0.187	0.245
LSI (10000)	0.207	0.284	0.298
NaiveMCTS (500)	0.116	0.197	0.250
NaiveMCTS (10000)	0.205	0.292	0.259
average	0.185	0.296	0.317

Table 3: Accuracy and gameplay strength of **AIM** models trained with different amount of training data.

Expected Loglikelihood				
Bot	10%	25%	50%	100%
WorkerRush	-0.908	-0.891	-0.878	-0.892
LSI (500)	-1.234	-1.230	-1.229	-1.229
LSI (10000)	-1.150	-1.510	-1.149	-1.146
NMCTS (500)	-1.215	-1.211	-1.205	-1.206
NMCTS (10000)	-1.154	-1.146	-1.148	-1.140
Gameplay Strength				
Bot	10%	25%	50%	100%
WorkerRush	0.547	0.553	0.568	0.518
LSI (500)	0.152	0.190	0.192	0.187
LSI (10000)	0.273	0.291	0.265	0.284
NMCTS (500)	0.102	0.138	0.188	0.197
NMCTS (10000)	0.294	0.209	0.216	0.292
average	0.274	0.277	0.286	0.296

that more training data might result in further small improvements in the model accuracy.

Experiment 4: Effect on the Branching Factor

A common explanation for why the policy network used by AlphaGO helps improve the performance of MCTS is that, in practice, it reduces the branching factor by assigning low probability values to a fraction of the possible actions. This section presents an analysis of the reduction in the branching factor that we get when using our proposed models. The left-most column in Table 4 shows the average number of unit-actions that a unit can perform in the game states reached when different bots play. As we can see, even if the theoretical maximum number of unit-actions in μRTS is 69, in average, a bot must select one among about 5 actions per unit. The right-most column in the table shows the average number of units in a game state (per player) so that we can have an idea of the number of possible player-actions (approximately n^m, where n is the average unit-actions per unit, and m is the average number of units).

The three middle columns show the average number of unit-actions that concentrate different percentages of the

Table 4: Branching factor (number of legal unit actions) in the states in the training data (100%), and the number of actions that result by sorting the actions by decreasing probability and keeping the ones that account for 99%, 90% or 50% of the probability mass.

Bot	100%	99%	90%	50%	Avg. units
fs1	5.02	3.75	3.54	1.90	19.49
fs2	5.02	3.91	3.33	1.77	
fs3	5.02	4.00	3.16	1.71	

probability mass, as predicted by the **AIM** models using different feature sets. For example, when using the feature set *fs2*, in average, if we just consider the first 3.33 unit-actions, we are already considering 90% of the probability mass. This means that, in practice, using an **AIM** model with feature set *fs2*, and an informed ϵ-greedy policy, we only need to consider about 66% ($\frac{3.33}{5.02} \approx 0.66$) of the unit-actions, and we would already be considering all of the most likely actions according to the model. As we can see, using different feature subsets achieves different reductions in branching factor, with feature set *fs3* cutting more drastically (except for the 99% column, which does not follow this trend for some reason we are still investigating).

In the final experiment below, we will evaluate the performance of the **AIM** models in the context of MCTS.

Experiment 5: Effect on MCTS

The last experiment is designed to see how the performance of the **AIM** model influences the performance of MCTS. Table 5 shows the gameplay performance for several MCTS configurations (always using a computation budget of 500 playouts per game cycle). We used the default configuration parameters of LSI, and the three parameters of NaiveMCTS set to $\epsilon_0 = 0.4$, $\epsilon_l = 0.33$ and $\epsilon_g = 0.0$). The first two rows show the baseline MCTS bots used for generating the training data, showing that they achieve a score of 0.788 and 0.837 respectively (remember the maximum score is 1.00, corresponding to winning every single game in every single map against every single opponent).

The next 9 rows of the table show the performance of *IN-MCTS*. In these 9 configurations, we use the **AIM** model learned from the *WorkerRush* as the playout policy (shown in our previous work to be the best for playouts), and we change the **AIM** model used in the tree policy. All these experiments used a computation budget of 500 playouts per game cycle. As we can see, surprisingly, the highest performance is achieved using feature set *fs1* (no features!). Our hypotheses is that given that naive sampling relies on ϵ-greedy, if an action has a very low probability according to the model, this action will never be selected regardless of the computation budget. Thus, models that cut the branching factor too much, might be performing pruning too aggressively. However, the model learned using *fs1*, still captures the overall distribution of actions, but at a coarser level, which seems to help guide MCTS, without pruning as aggressively, resulting in better performance. Comparing the performance achieved using *fs1*, we see that it is in the same

Table 5: Gameplay performance of IMCTS.

Bot	Score
Original bots	
LSI	0.788
NaiveMCTS	0.837
INMCTS (*fs1*)	
INMCTS(**AIM**(WR),**AIM**(WR))	0.935
INMCTS(**AIM**(WR),**AIM**(NMCTS5000))	0.946
INMCTS(**AIM**(WR),**AIM**(NMCTS10000))	0.937
INMCTS (*fs2*)	
INMCTS(**AIM**(WR),**AIM**(WR))	0.921
INMCTS(**AIM**(WR),**AIM**(NMCTS5000))	0.901
INMCTS(**AIM**(WR),**AIM**(NMCTS10000))	0.922
INMCTS (*fs3*)	
INMCTS(**AIM**(WR),**AIM**(WR))	0.893
INMCTS(**AIM**(WR),**AIM**(NMCTS5000))	0.894
INMCTS(**AIM**(WR),**AIM**(NMCTS10000))	0.897
NaiveMCTS (filtering actions)	
INMCTS(**AIM**(WR), no filter)	0.876
INMCTS(**AIM**(WR), 25% filter)	0.814

order or a bit higher than that achieved by *NaiveMCTS* when given a computation budget of 10000 playouts per game game cycle, which is remarkable.

An interesting question to be answered in future work is whether this is an anomaly observed because of the reliance of naive sampling on ϵ-greedy, or whether other bandit strategies would also suffer from this problem.

Finally, the bottom two rows show the performance achieved by using, as the unit probability model, a simple model that, selects a random set of actions (of a predefined size) and assigns them 0 probability, the remaining actions are given a uniform probability (this is to simulate the reduction in branching factor due to the models, but without actually using any kind of probability estimation). We tested both using no filtering of the actions (i.e., just using a uniform distribution), and filtering out 25% of the actions, which is about the amount of actions that were assigned probability values of near zero when using the *AIM* models. The results show that performance goes down with respect to not filtering, indicating that the performance of INMCTS comes not from just reducing the branching factor, but from the learned probability model.

Conclusions

This paper has presented a collection of experiments to understand the effect of different factors such as the training set, feature set and amount of training data have on the performance of action probability models. We also present results on how the performance of these action probability models affects the performance of MCTS.

Our results show that, when used in conjunction with our proposed informed naive sampling, probability models that discard too many of the possible actions seem to perform worse than those that perform a more coarse grained prediction. It is unclear to us at this point whether this is due to pruning the tree too aggressively, of because the models are pruning aggressively while not being all that accurate. As pat of our future work, we would like to study more accurate models (such as deep neural networks, as used in AlphaGO), and see if this effect can also be observed.

Also, we want to investigate the effect of incorporating probability models into other bandit policies, to see if the same trends are observed. In order to evaluate this, however, we must have in mind that policies like UCB1 cannot be applied directly to RTS games, since the large branching factor makes them perform very poorly. A possibility is modifying the way the MCTS is constructed (for example (Balla and Fern 2009) constructed a tree that only considered one unit per tree node, instead of the whole combinatorics of player-actions). Additionally, we are currently studying these concepts in the context of StarCraft, and seeing if the branching factor reduction that can be achieved is enough to help MCTS scale up to playing the full game of StarCraft.

References

Balla, R.-K., and Fern, A. 2009. UCT for tactical assault planning in real-time strategy games. In *Proceedings of IJCAI 2009*, 40–45.

Bennett, P. N. 2000. Assessing the calibration of naive bayes' posterior estimates. Technical Report CMU-CS-00-155, Carnegie Mellon University.

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

Buro, M. 2003. Real-time strategy games: a new AI research challenge. In *Proceedings of IJCAI 2003*, 1534–1535. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Ontanón, 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. *IEEE Transactions on Computational Intelligence and AI in Games (TCIAIG)* 5:1–19.

Ontanón, S. 2013. The combinatorial multi-armed bandit problem and its application to real-time strategy games. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*.

Ontañón, S. 2016. Informed monte carlo tree search for real-time strategy games. In *Proceedings of IEEE-CIG 2016*.

Rish, I. 2001. An empirical study of the naive bayes classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence*, volume 3, 41–46. IBM New York.

Shleyfman, A.; Komenda, A.; and Domshlak, C. 2014. On combinatorial actions and CMABs with linear side information. In *ECAI*, 825–830.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; et al. 2016. Mastering the game of go with deep neural networks and tree search. *Nature* 529(7587):484–489.

Uriarte, A., and Ontañón, S. 2016. Improving monte carlo tree search policies in starcraft via probabilistic models learned from replay data. In *Proceedings of AIIDE 2016*.