Informed Monte Carlo Tree Search for Real-Time Strategy games

Conference Paper · September 2016

DOI: 10.1109/CIG.2016.7860394

Santiago Ontanon
Drexel University
235 PUBLICATIONS 3,223 CITATIONS
SEE PROFILE

Informed Monte Carlo Tree Search for Real-Time Strategy Games

Santiago Ontañón
Computer Science Department, Drexel University
Philadelphia, PA, USA 19104
Email: santi@cs.drexel.edu

***Abstract*—The recent success of *AlphaGO* has shown that it is possible to combine machine learning with Monte Carlo Tree Search (MCTS) in order to improve performance in games with large branching factors. This paper explores the question of whether similar ideas can be applied to a genre of games with an even larger branching factor: *Real-Time Strategy* games. Specifically, this paper studies (1) the use of Bayesian models to estimate the probability distribution of actions played by a strong player, (2) the incorporation of such models into NaiveMCTS, a MCTS algorithm designed for games with combinatorial branching factors. We call this approach *informed* MCTS, since it exploits prior information about the game in the form of a probability distribution of actions. We evaluate its performance in the μRTS game simulator, significantly outperforming the previous state of the art.**

I. Introduction

The recent success of *AlphaGO* [1] has shown that it is possible to combine machine learning with Monte Carlo Tree Search (MCTS) [2] in order to improve performance in games with large branching factors. This paper explores the question of whether similar ideas can be applied to games with an even larger branching factor: *Real-Time Strategy* (RTS) games.

RTS games are a videogame genre where players command large armies in real time in order to defeat the other players. RTS games pose a significant challenge to artificial intelligence due to two main reasons: they have huge branching factors, and they are real-time, leaving very little time for players to decide which actions to play [3], [4]. Several approaches have been proposed to handle these games, such as MCTS for games with combinatorial branching factors [5], [6], portfolio approaches [7], [8], abstraction (simplify the game state and search in a simplified space) [9], [10], [11], hierarchical search [12], adversarial HTN planning [13], or case-based reasoning [14].

In this paper, we build upon work on MCTS approaches for RTS games, and study how to bring ideas from AlphaGO into this game genre. Specifically, we present:

- A collection of Bayesian models to estimate the probability distribution of actions played by a strong player, evaluating them by attempting to model a collection of bots in the context of an RTS game. These models are analogous to the *policy network* in AlphaGO.
- We incorporate these models into NaiveMCTS, a MCTS algorithm designed for games with combinatorial branching factors to inform the search process, resulting in what we call an *informed* MCTS approach. Our hypothesis is that the Bayesian models proposed in this paper (simpler than the deep neural networks used in AlphaGO) are enough to significantly improve MCTS.

We use the μRTS game simulator as our evaluation domain[1], and report results significantly outperforming the previous state of the art, LSI [5] and NaiveMCTS [6]. μRTS is deterministic, fully-observable and a forward model is available (necessary to implement game-tree search approaches).

The remainder of this paper is organized as follows. Section II introduces RTS games from an AI point of view. Section III presents our Bayesian probability distribution estimation models. Section IV describes how can they be incorporated into MCTS approaches, and finally Section V presents an empirical evaluation of the proposed approach.

II. Real-Time Strategy Games

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 [3]. Designing AI techniques for RTS games is challenging because:

- 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 [4] (for comparison, that of Chess is about 35, and that of Go about 180).
- They are *real-time*, which means that: 1) RTS 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, 2) players can issue actions simultaneously, and 3) actions are durative.

The reason for which the branching factor in RTS games is so large is that players controls 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)

[1]A fork of the μRTS project containing all the source code and datasets necessary to replicate the results reported in this paper can be downloaded from https://sites.google.com/site/santiagoontanonvillar/code/CIG-2016-microRTS-source-code.zip.

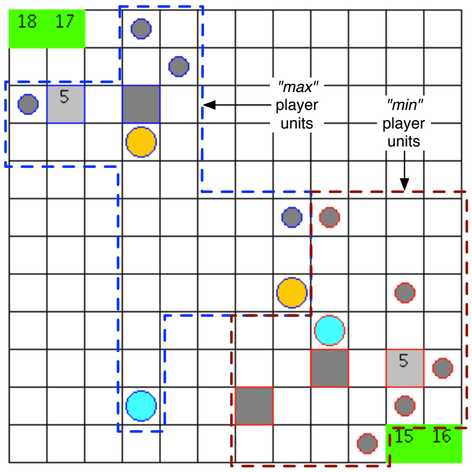

Fig. 1. A screenshot of the μRTS simulator. Square units correspond to "bases" (light grey, that can produce workers), "barracks" (dark grey, that can produce military units), and "resources mines" (green, from where workers can extract resources to produce more units), the circular units correspond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

and *min* (shown in red) control 9 units each. Consider the bottom-most circular unit 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.

In the remainder of this paper, we will use the following definition for an RTS game. An RTS game is 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.

III. Unit-Action Probability Estimation

AlphaGO used a deep neural network (DNN) to estimate the probability of each of the legal actions in a game state to be played by an expert player (the *policy network*). This network was trained from a large collection of expert games available on-line. A working hypotheses of this paper is that a simpler machine learning model than a DNN suffices to significantly improve the performance of MCTS. To assess such hypothesis, we propose simpler Bayesian models, based on the idea of the Naive Bayes classifier [15], and thus, with negligible training time. We present the proposed probability estimation models and how to generate training data below.

A. Bayesian Models

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 $\mathbf{x}(u, s) = \{x_1(u, s), ..., x_n(u, s)\}$, 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 action a that was chosen. The feature set used in our experiments is described in Section III-C. We experimented with two different models:

- **Calibrated Naive Bayes Model (CNB)**: while the Naive Bayes classifier [15] often works very well for classification purposes, it is well known that the probability distribution it estimates is not well "calibrated" [16], i.e., values tend to be very extremely close to either 0 or 1. Several approaches exist to better calibrate the posterior probability estimates of Naive Bayes classifiers, such as fitting them via a sigmoid function [16], or binning [17]. We propose a much simpler method, which achieved good results in our experiments: introducing a calibration parameter $\kappa > 0$ into the standard Naive Bayes formulation, as follows:

$$P(a|u, s) = \frac{1}{Z}\left(P(a)\prod_{i=1...n} P(x_i(u, s)|a)\right)^{\frac{1}{1+\kappa n}}$$

where n is the number of features used to represent the game state, $P(x_i(u, s)|a)$ and $P(a)$ are estimated from the training set[2], Z is just a normalization constant to make all the probabilities add up to 1, and κ is a calibration parameter, whose effect is to make the probability

[2]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 $P(a) = \frac{\text{number of times } a \text{ is selected}+1}{\text{size of the training set}+|A|}$

values less extreme. Thus, notice the **CNB** model is identical to the standard Naive Bayes formulation, except for the addition of the $\frac{1}{1+\kappa n}$ calibration exponent.
In our experiments, 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. Intuitively, when $\kappa = 0$, no correction is applied to the probabilities, and when κ grows, the probabilities grow less extreme. In the limit, when $\kappa \rightarrow \infty$, all probabilities would converge to the same number, making $P(a|u, s)$ the uniform distribution. In the particular case when $\kappa = 1$, the model corresponds to a geometric mean of the different factors in the probability estimation.

- **Action-Type Interdependence Model (AIM)**: the previous model does not consider that some actions might have a low or a high probability based on which other actions are legal. The AIM model captures the distribution given the actions that are legal in the current game state. Moreover, in order to reduce the number of parameters to estimate, 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, $c = \frac{1}{1+\kappa(n+|T|)}$, $T = legaltypes_u(u, s)$, and $F(a, s)$ is the product of the factors contributed by the features in $\mathbf{x}(s)$, as in the **CNB** model:

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

Finally, $L(type(a), T)$ is the product of a collection of new factors that estimate 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 estimated probability that an action of type t' was legal in a game state where an action of type t was selected.

In practice, we observed that 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[3]. Additionally, we also observed that adding feature selection also slightly improved the estimation. So, for the experiments reported in the remainder of this paper, we generated the probability models in the following way:

- For each unit type: we train a model using the subset of the training data referring to such unit type. If this subset is empty, then just train with 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).
- After training each model, we use a greedy additive wrapper feature selection method [18] to determine the subset of features from $\mathbf{x}(s)$ that maximize the predictive accuracy in the training set (by doing cross validation).
- After feature selection is performed, for each model, we find the parameter κ that maximizes the likelihood of the training set given the model.

B. Extending Unit-Action to Player-Action Distributions

When using the trained models to generate actions, it is necessary to actually generate player-actions, and not just unit-actions. When a player-action for player p needs to be generated in a game state s according to a unit-action distribution P (generated with either the **CNB** or the **AIM** models), we use the following procedure:

1) Push all the units that require unit-actions, $ready(p, s)$, 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 distributions, while respecting unit-action legality (L_u) and player-action legality (L_p). Moreover, notice that in practice, l will never be $\emptyset$ in step 4, since units in RTS games can always execute the *idle* action, which does not conflict with any other action. So, in practice this algorithm can sample a player-action without ever requiring going back to step 1. Also notice that choosing a random order in step 1 is necessary to prevent any undesired biases that can be caused by the order in which units are processed in steps 2 and 3 (since some unit-actions might prevent some other unit-actions).

C. Generating Training Data

Given that no available training data from expert players is available for μRTS, we generated training data in the following way. We selected the two current best reported Monte Carlo search-based bots in the literature (*LSI* [5], and *NaiveMCTS* [6]), and four hard-coded bots built into μRTS (*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

[3]The **CNB** model has learnable 6141 parameters, and **AIM** has 10971.

maps[4], 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 also experimented with the AHTN bot [13], but found that it only performed well in 2 of the 8 maps used in our evaluation, probably because the domain definition used was tailored to some specific type of maps.

Moreover, we repeated this round-robin tournament four times giving *NaiveMCTS* and *LSI* a computation budget of 500, 1000, 2000, and 5000 playouts per game frame respectively[5], resulting in four sets of game logs: RR500, RR1000, RR2000 and RR5000. We then constructed twelve datasets:

- I_{WR}: consisting of all the unit-actions of *WorkerRush* from RR500 (25739 instances).
- I_{LR}: consisting of all the unit-actions of *LightRush* from RR500 (38844 instances).
- I_{HR}: consisting of all the unit-actions of *HeavyRush* from RR500 (28321 instances).
- I_{RR}: consisting of all the unit-actions of *RangedRush* from RR500 (32753 instances).
- I^{500}_{LSI}, I^{1000}_{LSI}, I^{2000}_{LSI}, I^{5000}_{LSI}: consisting of all the unit-actions of *LSI* from RR500, RR1000, RR2000 and RR5000 respectively (89029, 74564, 70696, and 60000 instances).
- I^{500}_{nmcts}, I^{1000}_{nmcts}, I^{2000}_{nmcts}, I^{5000}_{nmcts}: consisting of all the unit-actions of *NaiveMCTS* from RR500, RR1000, RR2000 and RR5000 respectively (84784, 77200, 75462, and 70071 instances).

The feature vector $\mathbf{x}(u, s)$ used to represent each game state contains only 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 (or whether these cells are empty or are a wall). Adding more features could certainly improve performance, which will be part of our future work.

D. Empirical Evaluation

We evaluated the probability distribution models presented above in two different ways. By measuring how accurately can they predict the behavior of the bots (Table I), and by using them directly to play against the same six bots used to generate the training data (Table II).

a) Model Predictive Accuracy: Table I shows the predictive accuracy of both proposed models in each of the 12 datasets using a 10-fold cross validation. There is a total of 69 different actions unit can perform in μRTS, but each individual unit can perform only between 5 and 33 unit-actions. We measure both classification accuracy (1.0 corresponding to perfect predictions), and expected log-likelihood of the data. We can see that both models manage to predict the behavior of the four hard-coded bots very well (and specially the *Ranged Rush*, which the **CNB** model predicts with 0.846 accuracy and the **AIM** model with 0.892), but have a harder time predicting the behavior of the *LSI* and *NaiveMCTS* bots. Moreover, notice that the higher the computation budget, the easier *LSI* and *NaiveMCTS* are to predict, indicating that they might be converging to a more stable strategy. Moreover, the **AIM** model is systematically more accurate than the **CNB** model.

We also report the expected log-likelihood of the actions in the dataset given the model. This is a better metric to consider than classification accuracy given that we want to estimate the probability distribution of the actions, and not just predicting the most likely action. The best possible log-likelihood would be 0. As we can see, the trends follow exactly those for classification accuracy.

Figure 2 shows how the expected log-likelihood of the actions in the dataset change as a function of the calibration parameter κ for some of the datasets when a single **AIM** model is trained with all the data and without feature selection. As

TABLE I
CLASSIFICATION ACCURACY, AND AVERAGE LOG-LIKELIHOOD ACHIEVED WITH THE PROPOSED MODELS IN THE 12 DATASETS.

	CNB		**AIM**	
Dataset	*Acc.*	*Exp. l.l.*	*Acc.*	*Exp. l.l.*
I_{WR}	0.651	-0.952	0.679	-0.892
I_{LR}	0.800	-0.509	0.836	-0.441
I_{HR}	0.860	-0.381	0.877	-0.345
I_{RR}	0.846	-0.409	0.892	-0.318
I^{500}_{LSI}	0.380	-1.262	0.381	-1.229
I^{1000}_{LSI}	0.392	-1.237	0.397	-1.211
I^{2000}_{LSI}	0.410	-1.223	0.413	-1.193
I^{5000}_{LSI}	0.419	-1.211	0.425	-1.175
I^{500}_{nmcts}	0.393	-1.242	0.396	-1.206
I^{1000}_{nmcts}	0.407	-1.228	0.412	-1.193
I^{2000}_{nmcts}	0.418	-1.222	0.423	-1.181
I^{5000}_{nmcts}	0.429	-1.203	0.440	-1.160

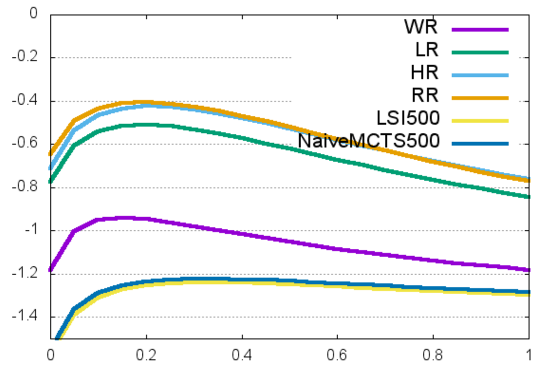

Fig. 2. Expected log-likelihood of the actions (vertical axis) in the training set for the **AIM** model using various training sets as a function of the κ calibration parameter (horizontal axis).

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

[5]Notice that "500 playouts per frame" does not mean "500 playouts per decision". Both LSI and NaiveMCTS spread computation across multiple frames when all units are busy in order to maximize playouts per decision.

we can see, introducing the calibration parameter improves the probability estimation significantly with respect to not having the calibration factor ($\kappa = 0$). In particular, values around the 0.1 to 0.3 range tend to obtain better results, but the maximum is achieved at a different value of κ depending on the training set. Finally, notice that the κ parameter does not have any effect on classification accuracy, since the relative order of which action has higher probability is not affected. κ only affects the probability distribution estimation, making the probabilities less extreme (i.e., less close to either 0 or 1).

b) Model Play Strength: Table II shows the average results of playing (by sampling from the trained distribution) in all eight maps described above, and against all the eight bots used to generate the training data. Each model played 20 games against each bot (10 as player 1, and 10 as player 2). Thus, each number in the table is the average of 160 games. Games longer than 3000 game frames were considered a draw. Bots scored 1 point for winning, and 0.5 for reaching a draw. We compare these results against those obtained by two baseline distributions: *Rnd*, which picks actions randomly, and *RndBiased*, where the *attack*, *harvest* and *return* actions have 5 times more probability than the rest of actions. The first thing we can see is that the **CNB** models perform much worse than the **AIM** models. For example, when training from the I_{WR} dataset, they achieve a score of 0.196 and 0.518 respectively in average. Models trained with the I_{WR} dataset achieved the highest score (mostly because *WorkerRush* employs a very aggressive strategy that works well against other, less aggressive, strategies). We also see that models trained from *LSI* and *NaiveMCTS* achieved better scores when trained from datasets generated with a higher computation budget. No model managed to win any game against the *WorkerRush* bot or *NaiveMCTS*, and only one bot won one game against *LSI*. So, although none of the trained models are strong enough to defeat the MCTS bots, they are significantly stronger than the baselines. The fact that the **AIM** model trained with the I_{WR} dataset can defeat most of the hardcoded bots is remarkable since: (1) it plays purely reactively, (2) the time required to train these models is negligible.

IV. 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 [2]. 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. Below, we describe *informed ϵ-Greedy sampling* and *informed naive sampling*, two informed policies, required to then explain *INMCTS*, our proposed algorithm.

A. Informed ϵ-Greedy Sampling

The *tree policy* of MCTS algorithms is usually defined as a *multi-armed bandit* (MAB) policy. A MAB is a problem where, given a predefined set of actions, an agent needs to select which actions to play, and in which sequence, in order to maximize the sum of rewards obtained when performing those actions. The agent has no information of the expected reward of each action initially, and needs to discover them by iteratively trying different actions. MAB policies are algorithms that tell the agent which action to select next, by balancing *exploration* (when to select new actions) and *exploitation* (when to re-visit actions that had already been tried in the past and looked promising).

MAB sampling policies traditionally assume that no a priori knowledge about how good each of the actions are exists. For example, UCT [19], one of the most common MCTS variants, uses the UCB1 [20] sampling policy, which assumes no a priori knowledge about the actions. A key idea used in AlphaGO is to employ a MAB policy that incorporated a prior distribution over the actions into a UCB1-style policy. Here, we apply the same idea to ϵ-greedy and then (in the next section) to Naive 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).

If P is uniform, this is equivalent to ϵ-greedy.

B. Informed Naive Sampling

Naive sampling is based on the idea of combinatorial multi-armed bandits (CMABs) [6] and internally uses a collection of ϵ-greedy sampling policies. Here we define *informed naive sampling*, as a result of replacing some of the internal ϵ-greedy sampling policies by informed ϵ-greedy.

Informed naive sampling takes four input parameters: a unit-action probability distribution P, and three constants ϵ_0, ϵ_l and ϵ_g, and determines which player-action to choose using a collection of MABs:

TABLE II
GAMEPLAY STRENGTH OF THE MODELS TRAINED WITH DIFFERENT DATASETS, COMPARED AGAINST TWO BASELINES (*Rnd*, AND *RndBiased*).

			CNB					**AIM**				
Bot	*Rnd*	*RndBiased*	I_{WR}	I^{500}_{LSI}	I^{5000}_{LSI}	I^{500}_{nmcts}	I^{5000}_{nmcts}	I_{WR}	I^{500}_{LSI}	I^{5000}_{LSI}	I^{500}_{nmcts}	I^{5000}_{nmcts}
Rnd	0.500	0.944	0.263	0.206	0.344	0.231	0.331	**1.000**	0.881	0.906	0.844	0.938
RndBiased	0.025	0.469	0.019	0.006	0.025	0.019	0.044	**0.881**	0.163	0.400	0.288	0.463
WorkerRush	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
LightRush	0.013	0.050	0.375	0.013	0.063	0.000	0.038	**0.650**	0.075	0.200	0.075	0.113
HeavyRush	0.038	0.063	0.488	0.038	0.138	0.038	0.075	**0.825**	0.213	0.313	0.200	0.300
RangedRush	0.038	0.050	0.425	0.025	0.088	0.025	0.100	**0.775**	0.163	0.250	0.125	0.263
LSI (500)	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**0.013**	0.000	0.000	0.000	0.000
NaiveMCTS (500)	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Average	0.077	0.197	0.196	0.036	0.082	0.039	0.073	**0.518**	0.187	0.259	0.197	0.259

- A set of *Local MABs*: For each unit $u_i \in ready(p, s)$, we define MAB_i that only considers the unit-actions of u_i.
- *Global MAB*: MAB_g considers each player-action that has been sampled so far. This means that in the first iteration, $t = 0$, MAB_g contains no player-actions at all.

Specifically, at each iteration t:

- With probability ϵ_0 choose *explore* and with probability $1 - \epsilon_0$ choose *exploit*:
  - If *explore* was selected: a legal player-action α^t is selected by using an *informed ϵ-greedy* policy (with ϵ_l) to select a unit-action for each unit $u_i \in ready(p, s)$ independently (i.e., the policy is used n times, one per local MAB), while ensuring the resulting player-action is legal. α^t is added to the global MAB.
  - If *exploit* was selected: a player-action α^t is selected by using an ϵ-greedy policy with ϵ_g over the player-actions already present in the global MAB.

Intuitively, when exploring, informed ϵ-greedy is used to select unit-actions that take into account both the prior probability distribution of unit-actions, and the current reward estimation of the unit-actions. When exploiting, the global MAB is used to find the player-action, among all the explored ones, with the maximum expected reward. After each iteration, the reward estimates of all the individual MABs are updated.

INMCTS (*informed NaiveMCTS*), is the result of replacing naive sampling from NaiveMCTS [6] by informed naive sampling.

V. EMPIRICAL EVALUATION

In order to evaluate the play strength of INMCTS, we performed four sets of experiments, reported in the following three subsections. All the experiments presented in this section are performed under the same conditions as those reported in Table II. The computation budget used for experiments was 500 playouts per cycle.

A. Experiment 1: Baselines

Table III shows the results of evaluating the gameplay strength of a collection of baseline algorithms against all the bots used for generating the training data, averaged over all the eight maps used in our experiments. The two left-most columns show the performance of NaiveMCTS, using two different *default policies*: *Random* and *RndBiased* (described in Section III-D). We can see that going from *Random* to *RndBiased* has a large effect in the gameplay strength, going from 0.709 for *Random* to 0.823 for *RndBiased*. The third column shows the performance of LSI, which is a bit lower than NaiveMCTS. Although previously reported results showed that LSI outperformed NaiveMCTS [5], in our experiments, that was the case for the smaller maps, while in the larger maps, when the branching factor grows drastically, NaiveMCTS outperforms LSI. This is analyzed later in Section V-D.

TABLE III
GAMEPLAY STRENGTH OF VARIOUS BASELINE CONFIGURATIONS AVERAGED OVER THE EIGHT MAPS USED IN OUR EXPERIMENTS (20 GAMES PER MATCH-UP PER MAP). [1] DEFAULT CONFIGURATION OF NAIVEMCTS AS ORIGINALLY REPORTED IN [6].

Algorithm	*NaiveMCTS*	*NaiveMCTS*[1]	LSI	*INMCTS*
Tree Policy	-	-	-	*RndBiased*
Default Policy	*Random*	*RndBiased*	*RndBiased*	*RndBiased*
Random	**1.000**	**1.000**	**1.000**	**1.000**
RndBiased	**1.000**	**1.000**	**1.000**	**1.000**
WorkerRush	0.600	**0.725**	**0.725**	0.700
LightRush	0.713	**0.900**	0.744	0.894
HeavyRush	**1.000**	**1.000**	**1.000**	**1.000**
RangedRush	0.700	0.788	0.725	**0.862**
LSI	0.363	0.669	0.500	**0.688**
NaiveMCTS	0.300	0.500	0.325	**0.613**
Average	0.709	0.823	0.752	**0.845**

The right-most column shows the performance of INMCTS when using *RndBiased* to inform both the *tree policy* and the *default policy*. The table shows that using *RndBiased* to inform the *tree policy* provides an additional performance improvement, going up to 0.845.

B. Experiment 2: Informed Sampling in the Tree Policy

Table IV shows results when we used *RndBiased* as the *default policy* in INMCTS, but we used different learned models to inform the *tree policy*. We do not show results with all the models shown in Table II due to space limitations, but we show some representative instances. The first thing we observe is that the **AIM** models outperform the **CNB** models. For example, when training models with the *WorkerRush* dataset (I_{RW}), we observed a performance of 0.813 with **CNB** versus 0.883 with **AIM**. Moreover, comparing Table IV with

TABLE IV
GAMEPLAY STRENGTH OF INMCTS USING DIFFERENT MODELS AS THE TREE POLICY (20 GAMES PER MATCH-UP PER MAP).

Algorithm	*INMCTS*	*INMCTS*	*INMCTS*	*INMCTS*
Tree Policy	**CNB**(I_{WR})	**AIM**(I_{WR})	**AIM**(I^{500}_{nmcts})	**AIM**(I^{5000}_{nmcts})
Default Plcy	*RndBiased*	*RndBiased*	*RndBiased*	*RndBiased*
Random	**1.000**	**1.000**	**1.000**	**1.000**
RndBiased	**1.000**	**1.000**	**1.000**	**1.000**
WorkerRush	0.700	**0.781**	0.713	0.748
LightRush	0.900	**0.975**	0.900	**0.975**
HeavyRush	**1.000**	**1.000**	**1.000**	**1.000**
RangedRush	0.863	**0.906**	0.838	0.875
LSI	0.594	**0.763**	0.638	0.675
NaiveMCTS	0.450	**0.644**	0.575	0.625
Average	0.813	**0.883**	0.833	0.861

TABLE V
GAMEPLAY STRENGTH OF INMCTS USING DIFFERENT PROBABILITY MODELS IN THE PLAYOUTS (20 GAMES PER MATCH-UP PER MAP).

Algorithm	*INMCTS*	*INMCTS*	*INMCTS*	*INMCTS*
Tree Policy	**AIM**(I_{WR})	**AIM**(I^{500}_{nmcts})	**AIM**(I^{5000}_{nmcts})	**AIM**(I^{5000}_{nmcts})
Default Plcy	**AIM**(I_{WR})	**AIM**(I^{500}_{nmcts})	**AIM**(I^{5000}_{nmcts})	**AIM**(I_{WR})
Random	**1.000**	**1.000**	**1.000**	**1.000**
RndBiased	**1.000**	**1.000**	**1.000**	**1.000**
WorkerRush	0.888	0.731	0.838	**0.950**
LightRush	**0.925**	0.675	0.763	**0.925**
HeavyRush	**1.000**	0.981	**1.000**	**1.000**
RangedRush	**0.963**	0.838	0.825	0.950
LSI	**0.844**	0.556	0.669	0.794
NaiveMCTS	**0.813**	0.450	0.513	0.650
Average	**0.929**	0.779	0.826	0.909

Table III we can see that informing the tree policy with the **AIM** model significantly outperforms NaiveMCTS.

The second effect we see is that when training models with the NaiveMCTS or LSI datasets, performance increases when training from datasets generated with a higher computation budget. For example, the performance of INMCTS when using an **AIM** model trained with I^{500}_{nmcts} is 0.833, and it goes up to 0.861 when trained with the I^{5000}_{nmcts} dataset.

Finally, we see that the best performance was achieved when using the **AIM**(I_{WR}) model (trained with the *WorkerRush* dataset), which outperformed models trained form NaiveMCTS or LSI. Our hypothesis is that this is because: 1) *WorkerRush* is a very aggressive early rush strategy, which works very well in the small maps used for testing; 2) *WorkerRush* is a deterministic strategy which is easy to model using our probability models; on the other hand, as can be seen from Table I, NaiveMCTS is harder to learn.

C. *Experiment 3: Probability models as Default Policies*

Table V shows results from similar experiments to the previous subsection, but where we used trained **AIM** models as the *default policy*. The three left-most columns show results when using the same model for both informing the *tree policy* and as *default policies*, where we can see that when using the model trained with the *WorkerRush* dataset, results improve significantly (to 0.929), but when using the models trained with NaiveMCTS data, performance goes down with respect

TABLE VI
MEDIAN, AVERAGE AND MAXIMUM BRANCHING FACTOR ENCOUNTERED IN EACH OF THE EIGHT MAPS.

	Branching		
Map	*Median*	*Average*	*Max*
1BW8x8	14.50	1466.35	2.65×10^5
2BW8x8	84.00	1.87×10^6	2.49×10^9
3BW8x8	106.00	4.53×10^5	1.75×10^8
4BW8x8	342.50	9.52×10^5	3.74×10^8
1BW12x12	30.50	1.23×10^6	6.14×10^8
2BW12x12	112.00	1.56×10^{12}	2.90×10^{15}
3BW12x12	–	–	–
4BW12x12	–	–	–

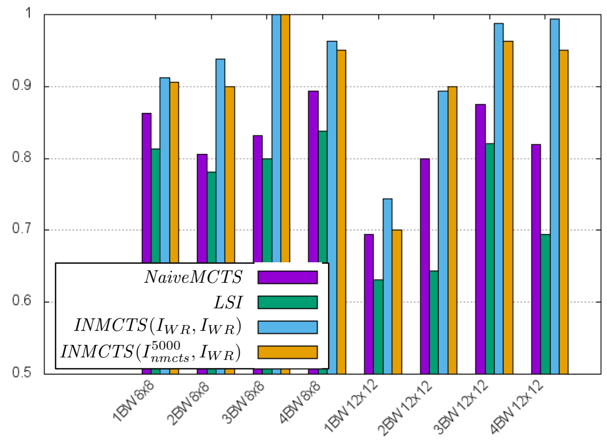

Fig. 3. Performance (vertical axis) in each of the 8 maps used in our experiments (horizontal axis) of two baselines, and the two INMCTS instantiations that performed better in our experiments, with computation budget 500.

to just using the *RndBiased* policy. The right-most column shows results when using a model trained with the *NaiveMCTS* dataset for informing the *tree policy*, and one trained with the *WorkerRush* dataset as the *default policy*, showing also very good performance (0.909).

In conclusion, we can see that the the idea of using informed sampling has the potential to significantly improve the performance of MCTS-based bots. The best combination reported in this paper (left-most column on Table V) has a win rate of over 80% against NaiveMCTS, and even higher against LSI. This is remarkable, since the probability distribution model we are using (**AIM**) is much simpler than the complex deep neural network (DNN) approach employed by AlphaGO. This confirms the hypothesis that simpler probability distribution models are enough to significantly improve the performance of MCTS. However, we still believe that performance can be increased even more by using better probability distribution models. In particular, more complex models, such as DNNs might be required when training models from complex bots such as NaiveMCTS or LSI, at the cost of increased computational cost during training.

D. *Effect of the Branching Factor*

Table VI shows the median, average and maximum branching factor encountered in each of the eight maps when making some of the baseline bots play against each other. As can be seen, the 12x12 maps result in significantly higher branching factors, and specially in the last two maps (where players start with 3 and 4 bases respectively), where branching factors were beyond the range we could calculate them exactly (but we could estimate some cases to be over 10^{18}).

Figure 3 shows the score achieved by two baseline models (NaiveMCTS and LSI) and the two best instantiations of INMCTS in our experiments in each of the eight maps. As can be seen, performance in each of the maps varies greatly. For example, performance in the first four maps (8x8 in size) is higher than in the second four maps (12x12 in size). We can also see that while LSI performs similar to NaiveMCTS in the first 4 maps, its performance drops significantly in the larger four maps. Moreover, in experiments where we increase the computational budget (to 5000 playouts per game frame, not reported in this paper), LSI actually outperforms NaiveMCTS in the first 4 maps, but still performs worse in the larger four maps (consistent with results reported by [5]).

Finally, all bots struggled to perform well in the fifth map (12x12 in size, and players starting with only one base and one worker). This is the map where the distance between the starting positions of the players is the largest. It is so large in fact, that the maximum length of the playouts used in our experiments (100 game frames) is not long enough to simulate a unit traveling all the way to the enemy base. Thus, MCTS-based bots behave erratically at the beginning of the game.

VI. Conclusions

This paper has explored the idea of informed sampling in the context of MCTS for games with combinatorial branching factors. The goal was to understand whether one of the key ideas that made AlphaGO surpass existing MCTS approaches can be also help in games with even larger branching factors. For that purpose, we proposed two probability distribution models, **CNB** and **AIM**, which play the role of the *policy network* of AlphaGO and were trained from datasets generated from existing bots. Our experimental results showed the proposed approach significantly outperforms the current state of the art.

Moreover, when modeling probability distributions from bots employing a simple strategy (such as *WorkerRush*), our probability estimation models are enough to significantly improve MCTS. In order to learn distributions from bots, or humans, employing more complex strategies (such as NaiveMCTS), a more complex probability model might be required.

As part of our future work, we would like to explore better probability distribution models, and compare also results against models trained using DNNs (which would likely require larger datasets to avoid overfitting). Moreover, in this paper we did not consider other components of AlphaGO, such as the value network, or learning by self-play, which we would like to study in the future. We would also like to study the scalability of the approach with respect to the game definition. For example, the number of parameters to be learned from data grows linearly with the number of possible unit-actions for the **CNB** model, but quadratically for the **AIM** model. Variables such as "map size" affect the MCTS search process, but not the probability distribution model learning process. Finally, we believe the idea of informed sampling can be generalized to many other MCTS algorithms for other domains. We have already started work on these ideas on StarCraft, where the main new challenges are the fact that the game is partially observable, and that we do not have a forward model.

References

[1] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[2] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 4, no. 1, pp. 1–43, 2012.

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

[4] S. Ontanón, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill, and M. Preuss, "A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft," *IEEE Transactions on Computational Intelligence and AI in Games (TCIAIG)*, vol. 5, pp. 1–19, 2013.

[5] A. Shleyfman, A. Komenda, and C. Domshlak, "On combinatorial actions and CMABs with linear side information," in *ECAI*, 2014, pp. 825–830.

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

[7] D. Churchill and M. Buro, "Portfolio greedy search and simulation for large-scale combat in starcraft," in *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*. IEEE, 2013, pp. 1–8.

[8] N. Justesen, B. Tillman, J. Togelius, and S. Risi, "Script-and cluster-based UCT for starcraft," in *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*. IEEE, 2014, pp. 1–8.

[9] A. Uriarte and S. Ontañón, "Game-tree search over high-level game states in RTS games," in *Tenth Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2014)*, 2014.

[10] D. Soemers, "Tactical planning using mcts in the game of starcraft," Ph.D. dissertation, Masters thesis, Department of Knowledge Engineering, Maastricht University, 2014.

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

[12] M. Stanescu, N. A. Barriga, and M. Buro, "Hierarchical adversarial search applied to real-time strategy games." in *Tenth Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2014)*, 2014.

[13] S. Ontañón and M. Buro, "Adversarial hierarchical-task network planning for complex real-time games," in *Proceedings of the 24th International Conference on Artificial Intelligence*. AAAI Press, 2015, pp. 1652–1658.

[14] D. Aha, M. Molineaux, and M. Ponsen, "Learning to win: Case-based plan selection in a real-time strategy game," in *ICCBR'2005*, ser. LNCS, no. 3620. Springer-Verlag, 2005, pp. 5–20.

[15] I. Rish, "An empirical study of the naive bayes classifier," in *IJCAI 2001 workshop on empirical methods in artificial intelligence*, vol. 3. IBM New York, 2001, pp. 41–46.

[16] P. N. Bennett, "Assessing the calibration of naive bayes' posterior estimates," Carnegie Mellon University, Tech. Rep. CMU-CS-00-155, 2000.

[17] B. Zadrozny and C. Elkan, "Obtaining calibrated probability estimates from decision trees and naive bayesian classifiers," in *ICML*, vol. 1. Citeseer, 2001, pp. 609–616.

[18] R. Kohavi and G. H. John, "Wrappers for feature subset selection," *Artificial intelligence*, vol. 97, no. 1, pp. 273–324, 1997.

[19] L. Kocsis and C. Szepesvri, "Bandit based monte-carlo planning," in *Proceedings of ECML 2006*. Springer, 2006, pp. 282–293.

[20] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, no. 2, pp. 235–256, 2002.