Proceedings, The Twelfth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-16)

Improving Monte Carlo Tree Search Policies in StarCraft via Probabilistic Models Learned from Replay Data

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

Abstract

Applying game-tree search techniques to RTS games poses a significant challenge, given the large branching factors involved. This paper studies an approach to incorporate knowledge learned offline from game replays to guide the search process. Specifically, we propose to learn Naive Bayesian models predicting the probability of action execution in different game states, and use them to inform the search process of Monte Carlo Tree Search. We evaluate the effect of incorporating these models into several Multiarmed Bandit policies for MCTS in the context of STARCRAFT, showing a significant improvement in gameplay performance.

Introduction

Real-Time Strategy (RTS) games provide a popular and challenging domain for research in Artificial Intelligence (AI) (Buro 2003). One of the reasons RTS games are hard is because the branching factor they involve is very large (Ontañón et al. 2013). Even Monte Carlo tree search (MCTS) approaches (used successfully in complex games like Go) do not scale well. Past approaches to this problem have explored abstractions (Kovarsky and Buro 2005; Uriarte and Ontañón 2014), portfolios (Churchill and Buro 2013; Barriga, Stanescu, and Buro 2015), hierarchical search (Stanescu, Barriga, and Buro 2014), or a combination of the previous ones (Ontañón and Buro 2015).

In this paper we explore an approach to improve the performance of MCTS approaches in STARCRAFT based on modeling the behavior of human experts. Specifically, we present a supervised probabilistic model of squad unit behavior, and show how to train this model from human replay data. This model captures the probability with which humans perform different actions in different game circumstances. We incorporate this model into the policies of a MCTS framework for STARCRAFT to inform both the *tree policy* and the *default policy* significantly outperforming a baseline MCTS approach.

The remainder of this paper is organized as follows. First, we provide some background in the context of RTS games and MCTS. Then we introduce a methodology to capture the behavior of human experts and how to apply it into different policies of a MCTS framework. Finally, we present our empirical evaluation in STARCRAFT (a popular RTS game).

Background

RTS games are a sub-genre of strategy games where players need to build an economy (gathering resources and building a base) and military power (training units and researching technologies) in order to defeat their opponents (destroying their army and base). The challenges in RTS games not present in other traditional board games are: they are *simultaneous move* games (more than one player can issue actions at the same time), they have *durative actions* (actions are not instantaneous), they are *real-time* (each player has a very small amount of time to decide the next move), they are *partially observable* (players can only see the part of the map that has been explored), they might be *non-deterministic* (some actions have different outcomes), and they have a *large search space* (the number of possible board configurations). However, in this paper, we will only consider fully observable settings.

In this paper we focus on techniques to mitigate the large branching factor in games like STARCRAFT (showed to be between 10^{50} and 10^{200} when units can be controlled simultaneously (Ontañón et al. 2013)). As a reference, the branching factor of Go is around 250. Specifically, we present some enhancements that can be applied in MCTS policies.

The main concept of MCTS is that the value of a state may be approximated using repeated stochastic simulations from the given state until a terminal state (or until a termination condition is met) (Browne et al. 2012). During search, MCTS employs two different policies:

- The *tree policy* is used to determine which node to expand next in the search tree and balances the *exploration* (look in areas that have not been explored yet) and *exploitation* (look at the most promising areas of the tree);
- the *default policy* used to simulate games until a terminal node is reached. The simplest default policy that can be used to run the simulations is selecting uniformly random actions for each player.

The main idea of this paper is to inform these policies with prior knowledge to bias the search towards the most probable (and promising) areas. This idea has already been applied with success in other games like Poker or Go. In Poker, Ponsen et al. (2010) learned an opponent model to bias the *tree policy*. Coulom (2007) calculated the Elo rating in Go to inform both policies (tree and default), Gelly and Silver (2007) experimented with combining offline reinforcement learning knowledge and online sample-base search knowledge; this idea was further explored later in AlphaGo (Silver et al. 2016). In our previous work, we showed this can also improve small-scale RTS gameplay (Ontañón 2016). In this paper, we show how this idea can be scaled up to large RTS games, such as StarCraft, and how can replay data be used to generate the required training set.

Modeling Squad Behavior in StarCraft

The main idea of our approach is to add offline knowledge from human experts into MCTS. In order to do that, we use an abstract representation of the game state to group units into "squads", then we define a squad-action probability model, and we show how to learn it from data.

Game State Abstraction

We use the same abstraction proposed by Uriarte and Ontañón (2014), where the map is represented as a graph where each node corresponds to a region (the region decomposition is performed by the library BroodWar Terrain Analyzer BWTA (Perkins 2010)), and edges represent adjacent regions. Additionally, all the units of the same type inside each region are grouped together into "squads"; and each "squad" has the following information: *Player* (owner of the squad), *Type* (type of units in the squad), *Size* (number of units), *Average HP* (average hit points of all units in the squad), *Region* (which region is this squad in), *Action* (which action is this squad currently performing), *Target* (the target region of the action, if applicable), and *End* (in which game frame is the action estimated to finish).

The possible actions that a "squad" can perform are: *Move* (to an adjacent region), *Attack* (all the enemies in the current region), and *Idle* (do nothing during 400 frames). This abstraction has been observed to reduce the branching factor of states that using the raw game state of StarCraft have a branching factor of the order of 10^{300} to 10^{10} (Uriarte and Ontañón 2014).

We extend this abstraction by distinguishing between different types of *Move* actions. Specifically, for each *Move* action, we define the following set of boolean features depending on the properties of the target region to move:

- **To Friend**: whether the target region has friendly units.
- **To Enemy**: whether the target regions has enemy units.
- **Towards Friend**: whether the target region is closer to a friendly base than the current one.
- **Towards Enemy**: whether the target region is closer to a enemy base than the current one.

Notice that these features are not mutually exclusive, i.e., all 16 combinations are possible.

Squad-Action Naive Bayes Model

This model captures the probability distribution by which human experts perform each of the given actions in a given situation and given the actions that are legal for a given squad. As defined in the previous subsection, the set of all possible action types T a squad can perform (*Idle*, *Attack* and *Move* (toFriend, toEnemy, towardsFriend, towardsEnemy) contains 18 different action types (we distinguish *Move* actions by their features (described above), since there are 2^4 different *Move* actions, the total number of different action types a squad can perform is $2 + 2^4 = 18$). Moreover, in a given game situation the set of actions a squad can perform is bounded by the number of adjacent regions to the region at hand, and there might be more than one squad action with the same action type (e.g., there might be more than one adjacent region characterized by the same move features). Let us define $\mathcal{T} = \{t_1, ..., t_{18}\}$ to be the set of action types that squads can perform, and let $\mathcal{A}_s = \{a_1, ..., a_n\}$ to be the set of squad actions a squad can perform in a given game state s (we call this the set of *legal* actions), where we write the type of an action as $type(a) \in T$. Now, let $X_1, ..., X_{18}$ be a set of Boolean variables, where X_i represents whether in the current state s, the squad at hand can perform an action of type t_i. Moreover, variable T denotes the type of the action selected by the squad in the given state ($T \in \mathcal{T}$), and A denotes the actual action a squad will select ($A \in \mathcal{A}_s$). Then, we will use the conditional independence assumption, usually made by Naive Bayes classifiers, that each variable X_j is conditionally independent of any other variable X_i given T, to obtain the following model:

$$P(T|X_1, \dots, X_n) = \frac{1}{Z} P(T) \prod_{j=1}^{n} P(X_j|T)$$

where Z is a normalization factor, $P(T = t_i)$ is the probability of a squad to choose action of type t_i given that $X_i = true$, and $P(X_j|T)$ is the probability distribution of feature X_j, given T. Below we will show how to estimate these probability distributions from replay data.

Finally, since there might be more than one action in $\mathcal{A}_s$ with the same given action type (e.g., there might be two regions a squad can move to characterized by the same features), we calculate the probability of each action as:

$$P(A|X_1, \dots, X_n) = \frac{P(type(A)|X_1, \dots, X_n)}{|\{a \in \mathcal{A}_s | type(a) = type(A)\}|}$$

Training Data

To be able to learn the parameters required by the model, we need a dataset of squad actions. We extracted this information from professional human replays. Extracting information from replay logs has been done previously (Weber and Mateas 2009; Synnaeve and Bessière 2012; Uriarte and Ontañón 2015). To generate the required dataset we built a parser that, for each replay, proceeds as follows[1]:

[1]Source code of our replay analyzer, along with our dataset extracted from replays, can be found at https://bitbucket.org/auriarte/bwrepdump

- First, at each time instant of the replay, all units with the same type in the same region are grouped into a squad.
- Second, for each region with a squad, the set of legal actions is computed.
- Third, the action that each unit is performing is transformed to one of the proposed high-level actions (if possible). For example, low-level actions like *Move*, *EnterTransport*, *Follow*, *ResetCollision* or *Patrol* become *Move*. In the case of the high-level action *Move*, the target region is analyzed to add the region features to the action.
- Fourth, the most frequent action in each squad is considered to be the action that squad is executing.
- Finally, for each squad and each time instant in a replay, we compute its squad state as $s = (g, r, t, T, A)$ where g is the unit's type of the squad, r is the current region of the squad, t is the type of the action selected, T is the set of types of the legal actions at region r, and A is the actual set of legal actions at region r.

Each time that the state of a squad changes in a replay (or a new squad is created), we record it in our dataset, and it constitutes one of the training instances.

Once we have the training set $S = \{s_1, \ldots, s_m\}$, we can use the Maximum Likelihood Estimation (MLE) to estimate the parameters of our model as:

$$P(T = t) = \frac{|\{s \in S | s.t = t\}|}{|\{s \in S | t \in s.T\}|}$$

$$P(X_j = true | T = t) = \frac{|\{s \in S | s.t = t \wedge t_j \in s.T\}|}{|\{s \in S | s.t = t\}|}$$

Informed Monte Carlo Tree Search

We incorporated the model described above into MCTS, a family of planning algorithms based on sampling the decision space rather than exploring it systematically. As described above, MCTS employs two different *policies* to guide the search: a *tree policy* and a *default policy*. The squad-action probability model learned above can be used in MCTS in both policies.

Moreover, while a squad-action probability model can be used directly as a *default policy*, to be used as a *tree policy*, it needs to be incorporated into a multi-armed bandit policy.

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 (Kocsis and Szepesvári 2006), one of the most common MCTS variants, uses the UCB1 (Auer, Cesa-Bianchi, and Fischer 2002) 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.

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

Given $0 \leq \epsilon \leq 1$, 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 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).

When P is the uniform distribution, this is equivalent to the standard ϵ-greedy policy. We will use the acronym *NB-ϵ* to denote the specific instantiation of informed ϵ-greedy when using our proposed Naive Bayes model to generate the probability distribution P.

Best Informed ϵ-Greedy Sampling

Best Informed ϵ-Greedy Sampling is a modification over the previous MAB policy, that treats the very first iteration of the MAB as a special case, Specifically, at each iteration t, action a_t is chosen as follows:

- If $t = 0$, choose the most probable action a_t given the probability distribution P: $a_t = argmax_{a \in A} P(a)$.
- Else, use regular Informed ϵ-Greedy Sampling.

Our experiments show that this alternative MAB policy is very useful in cases where we have a very small computation budget (e.g., in the deeper depths of the MCTS tree), and thus, seems very appropriate to real-time games.

We will use the acronym *BestNB-ϵ* to denote the specific instantiation of best informed ϵ-greedy when using our proposed Naive Bayes model to generate the distribution P.

Other MAB Sampling Policies

The idea of incorporating predictors (probabilistic or not) has been explored in the past in the context of many other MAB sampling policies. For example, PUCB (Predictor + UCB) (Rosin 2010) is a modification of the standard UCB1 policy incorporating weights for each action as provided by an external predictor (such as the one proposed in this paper). Chaslot et al. (2007) proposed two *progressive strategies* that incorporate a heuristic function over the set of actions that is taken into account during sampling. Another example is the sampling policy used by AlphaGO (Silver et al. 2016), which is related to both progressive strategies.

One problem of UCB-based policies is that they require sampling each action at least once. In our setting, however, there might be nodes in the MCTS tree with a branching factor larger than the computational budget, making those policies inapplicable. An exception is PUCB, which is designed for not having to sample each action once. We experimented with PUCB in our application domain with very poor results. However, since even if PUCB does not require to sample all the actions once, it still requires to reevaluate the value of each action in order to find the action that maximizes this value. Given the large branching factor in our domain, this resulted in impractical execution times, which made us only consider ϵ-greedy-based strategies. As explained below, as part of our future work, we would like to explore additional policies taking into account the particularities of our domain.

Informed MCTSCD in StarCraft

In order to incorporate our informed MCTS approach into an actual StarCraft playing bot, we followed the same steps described in (Uriarte and Ontañón 2014). To do so, we need to: (1) define a mapping between low-level StarCraft states and high-level states using the game state and action abstraction presented at the beginning of this paper; (2) use a MCTS algorithm that can handle **durative actions** and **simultaneous moves** (we used MCTSCD (Uriarte and Ontañón 2014)), (3) use our proposed *squad-action Naive Bayes model* to inform the default and tree policies in MCTSCD; (4) map the best high-level action selected back to a low-level action.

Concerning mapping low-level states to high-level states and actions, most StarCraft bots are decomposed in several individual agents that perform different tasks in the game, such as scouting or construction (Ontañón et al. 2013). One of such agents is typically in charge of combat units, and is in charge of controlling a military hierarchy architecture. This agent usually uses the intermediate concept of *squads* to control groups of units. However, this agent might group units in a different way that our desired high-level abstraction. Therefore we need to map each unit's agent group to its high-level squad. Notice that this mapping might not be one to one since a group of units crossing a chokepoint will be split in two abstract squads (one for each region); and groups with a mix of unit types will be split in one abstract squad for each unit type.

Once we have a high-level game state, we use an *informed MCTSCD* to search the beast action for each group. Informed MCTSCD is an extension of MCTSCD that uses our presented *squad-action Naive Bayes model* to inform the policies (tree and default). Since RTS games are real-time, we perform a search process periodically, and after each search, the action associated with each squad is updated with the result of the search.

Experimental Evaluation

In order to evaluate the performance of informed MCTSCD we performed a set of experiments using our StarCraft bot (Uriarte and Ontañón 2012) that uses the proposed informed MCTSCD to command the army during a real game.

Experimental Setup

Dealing with partial observability, due the *fog of war* in StarCraft, is out of the scope of this paper. Therefore, we disabled fog of war in order to have perfect information of the game. We also limited the length of a game to avoid situations where bots are unable to win because they cannot find all the opponent's units (StarCraft ends when all the opponent's buildings are destroyed). In the StarCraft AI competition the average game length is about 21,600 frames (15 minutes), and usually the resources of the initial base are gone after 26,000 frames (18 minutes). Therefore, we decided to limit the games to 20 minutes (28,800 frames). If we reach the timeout we consider the game a tie.

In our experiments, we performed one call to informed MCTSCD every 400 frames, and pause the game while the search is taking place for experimentation purposes.

Informed MCTSCD has several parameters which we set as follows: for any policy using an ϵ-greedy component ϵ is set to 0.2; to decide the player in a *simultaneous node* we use an *Alt* policy (Churchill, Saffidine, and Buro 2012) that alternate players; the *length of playouts* (or simulations) is limited to unfold until 2,880 frames (2 minutes of gameplay; this number is extracted from an empirical evaluation described in the next subsection) or until a terminal node is reached (and with a general timeout of 28,800 frames); and as a forward model for playouts (or "simulator") we use the *Decreasing DPF model* (where the DPF of an army is decreased every time a unit is killed) using a learned Borda Count target selection policy (Uriarte and Ontañón 2016). We experimented with executing informed MCTSCD with a computational budge from 1 to 10,000 playouts; and with the following configurations of (*tree policy*, *default policy*):

- **(ϵ, UNIFORM)**. Our baseline using an ϵ-greedy for the tree policy, and a uniform random default policy.
- **(ϵ, NB)**. Same as previous but changing the default policy to our proposed *Squad-Action Naive Bayes Model*.
- **(NB-ϵ, NB)**. An *informed ϵ-greedy sampling*, that uses our proposed *Squad-Action Naive Bayes Model* to generate the probability distribution P, for the tree policy.
- **(BestNB-ϵ, NB)**. For this configuration we changed the tree policy to use a *best informed ϵ-greedy sampling* with the *Squad-Action Naive Bayes Model*.

We used the StarCraft tournament map Benzene for our evaluation and we ran 100 games with our bot playing the Terran race against the built-in Terran AI of StarCraft that has several scripted behaviors chosen randomly at the beginning of the game.

Results

In our first experiment we evaluated the performance of MCTSCD with *playouts* (or simulations) of different durations. The computational budget of MCTSCD was fixed to 1,000 playouts, and we used standard ϵ-greedy for the tree policy and a uniform distribution as the default policy.

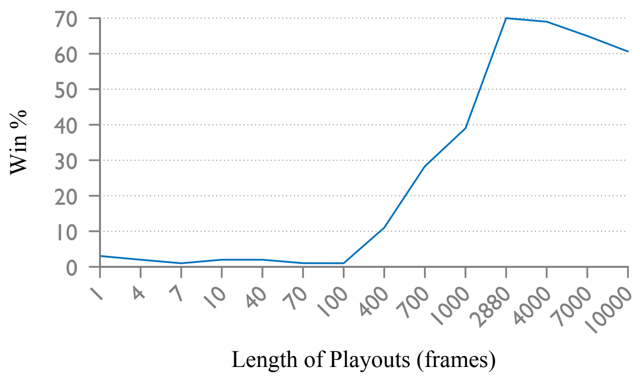

Figure 1: Win % using a MCTSCD with an ϵ-greedy tree policy and a uniform random default policy.

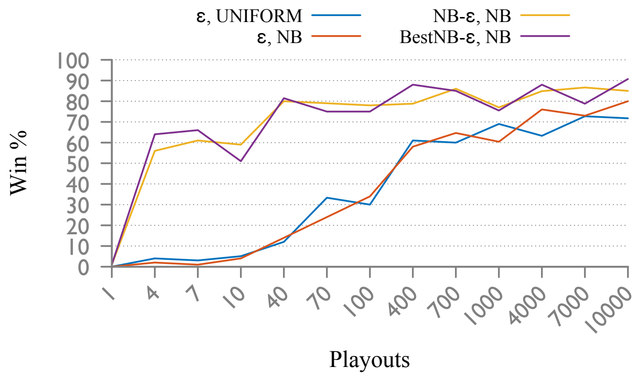

Figure 2: Comparison of Win % using a MCTSCD with different policies (tree policy, default policy).

Intuitively, increasing the length of playouts, increases the lookahead of the search. As we can observe in Figure 1, performance is very low if playout length is kept below 100 frames. It increases exponentially between 100 up to 2,880 frames (from $\approx$ 4 seconds to 2 minutes), and after that the performance start to degrade. Our hypotheses concerning the performance degradation after 2,880 frames is because of the inaccuracies in our forward model. For example, the forward model used, does not simulate production, so after 2 minutes of playout simulation, the resulting state would be probably different from the actual state the game will be in after 2 minutes given that no new units are spawned during playouts. Secondly, our forward model is only an approximation, and the slight inaccuracies in each simulation over a chain of approximate combat simulations can compound to result into very inaccurate simulations.

Figure 2 shows the win % (i.e., wins without reaching the timeout) using several tree and default policies for different computation budgets. Starting from the extreme case of running MCTSCD with a single playout (which basically would make the agent just play according to the tree policy, since the first child selected of the root node will be the only one in the tree, and thus the move to be played), all the way to running 10,000 playouts. Playout length was set

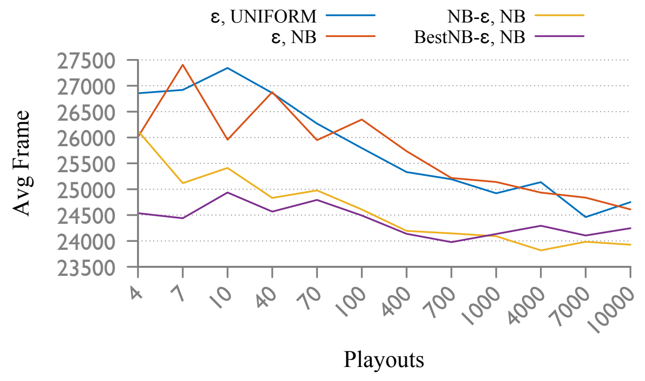

Figure 3: Comparison of average frames it took to win a game using a MCTSCD with different policies (tree policy, default policy). Lower is better.

to 2,880 frames. As expected, as the number of playouts increases, performance also increases. Moreover, we can observe a significant difference between using a standard ϵ-greedy tree policy (ϵ), which achieves a win % of about 70% when using 10,000 playouts, and using an informed tree policy (BestNB-ϵ or NB-ϵ), which achieve a win % of 90% and 85% respectively. Additionally, using informed tree policies, we reach a win % of about 80% by using as few as just 40 playouts (which is a surprisingly low number!). This shows that the probability distribution captured by the Naive Bayes model can significantly help MCTSCD during the search process in guiding the search toward promising areas of the search tree. On the other hand, the performance difference between using an informed default policy or not (*NB* vs *UNIFORM*) is not very large.

Figure 3 shows the average amount of time (in game frames) that our approach took to defeat the opponent, showing again a clear advantage for informed policies. Figure 4 shows the average *kill score* achieved by the opponent at the end of the game. Since this captures how many units our MCTSCD approach lost, lower is better. Again we see a clear advantage of informed strategies, but this time only for small number of playouts.

Finally, we analyzed the computation time required by each configuration, since we are targeting a real-time environment. Figure 5 shows how the increment of playouts leads to a linear time growth. And the *UNIFORM* default policy is faster. Mainly due the fact that it has lower chances to engage combats, which are expensive to simulate using our forward model (this is because, when there is a combat, our forward model needs to simulate the attacks of all the units involved, which requires more CPU time than when units just move around in squads).

In summary, we can see that adding the Naive Bayes model learned offline into MCTSCD improves the performance significantly. Of particular interest for RTS games is the fact that performance is really good with a very small number of playouts, since the model can guide MCTS down the branches that are most likely to be good moves. Using less than 100 playouts in any of our informed scenarios is

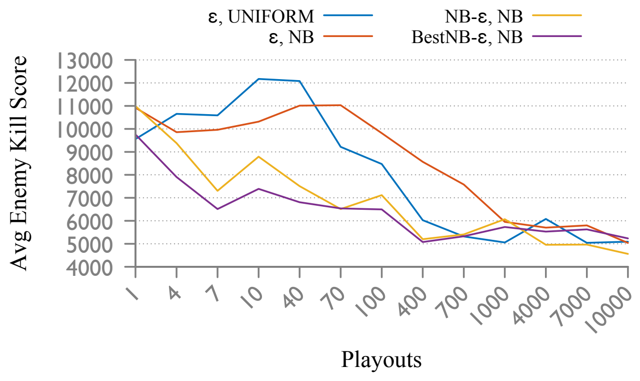

Figure 4: Comparison of average enemy's kill score using a MCTSCD with different policies (tree policy, default policy). Lower is better.

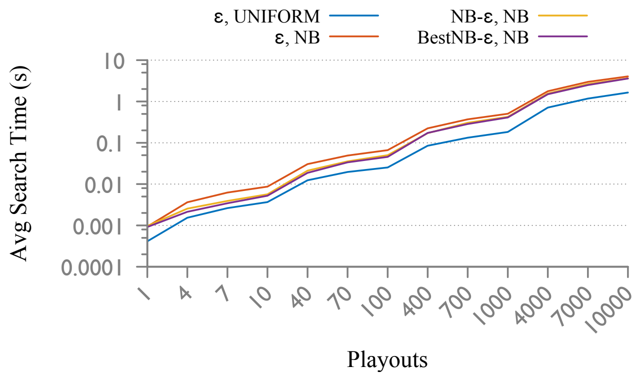

Figure 5: Comparison of average search time using MCTSCD with different tree and default policies.

enough to match the performance of MCTSCS with 10,000 playouts. Notice also that the search time with 100 playouts is less than 0.1 seconds. Suggesting that this approach could be applied without pausing the game during the searches. Moreover, we would like to point out (as shown in Figures 6 and 7) that the remaining 10% - 15% of games that are not shown as wins in Figure 2 for BestNB-ϵ or NB-ϵ, are actually not loses, but ties, and most of those are ties because our system defeated the enemy but was unable to find the last buildings. This was due to a limitation in our abstraction, where if a region is too large, MCTSCD does not have the capability of asking a unit to explore the whole region (since for MCTSCD that whole region is a single node in the map graph). As part of our future work, we would like to improve our map abstraction for not including regions that are larger than the average visibility range of units, in order to prevent this from happening.

Conclusions

This paper experiments with the idea of incorporating offline knowledge into MCTS for RTS games. Specifically, we proposed a Bayesian *squad-action probability distribution* model trained from replay data to capture the behavior

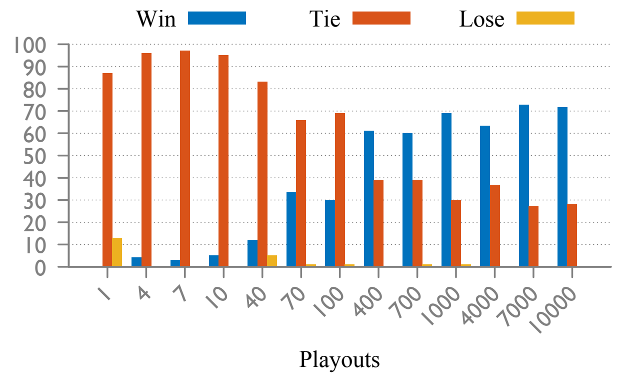

Figure 6: Win/Tie/Lose % of MCTSCD(ϵ,UNIFORM).

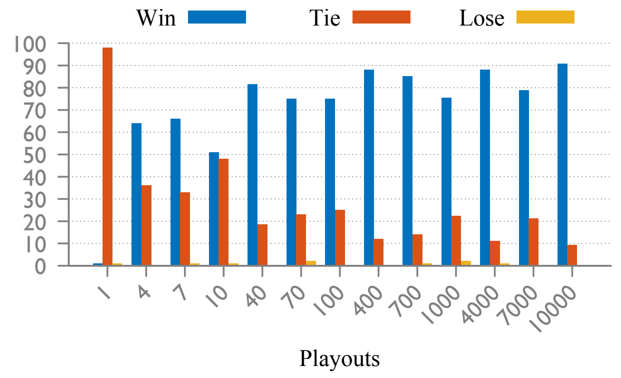

Figure 7: Win/Tie/Lose % of MCTSCD(BestNB-ϵ,NB).

of expert players in controlling squads in the game. We then used this model to inform the *tree policy* and the *default policy* of a MCTSCD algorithm.

Our results show that informing MCTS with out squad-action probability distribution model results in a great improvement in performance, specially when using it in the *tree policy* and when we have a tight computation budget. We saw that NB-ϵ and BestNB-ϵ policies achieve a significantly higher win ratio than standard ϵ-greedy. Additionally, BestNB-ϵ wins in less time and loses less units than NB-ϵ.

As part of our future work we would like to explore incorporating our model into a wider range of sampling strategies, such as NaïveMCTS, which is designed to deal with a combinatorial MAB like the one we are facing in RTS games. We would also like to explore the idea of using online knowledge to model the behavior of the current opponent (i.e., refine the probability model with the behavior we observe of the current player). Also, our current MCTS framework only considers military units, we would like to extend our framework to economy actions in order to have MCTS take control of all the units of the game, and not just the military units as we are doing now. Finally, we want to incorporate strategies to deal with partial observability to be able to handle partially observable games (i.e., fog of war).

References

Auer, P.; Cesa-Bianchi, N.; and Fischer, P. 2002. Finite-time analysis of the multiarmed bandit problem. *Machine learning* 47(2-3):235–256.

Barriga, N. A.; Stanescu, M.; and Buro, M. 2015. Puppet search: Enhancing scripted behavior by look-ahead search with applications to real-time strategy games. In *AIIDE*.

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. *TCIAIG* 4(1):1–43.

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

Chaslot, G. M. J.; Winands, M. H.; Herik, H. J. v. d.; Uiterwijk, J. W.; and Bouzy, B. 2007. Progressive strategies for Monte-Carlo tree search. In *JCIS*.

Churchill, D., and Buro, M. 2013. Portfolio greedy search and simulation for large-scale combat in StarCraft. In *CIG*, 1–8. IEEE.

Churchill, D.; Saffidine, A.; and Buro, M. 2012. Fast heuristic search for RTS game combat scenarios. In *AIIDE*. AAAI Press.

Coulom, R. 2007. Computing Elo ratings of move patterns in the game of Go. *ICGA* 30(4):198–208.

Gelly, S., and Silver, D. 2007. Combining online and offline knowledge in UCT. In *ICML*, 273–280.

Kocsis, L., and Szepesvári, C. 2006. Bandit based monte-carlo planning. In *ECML*, 282–293.

Kovarsky, A., and Buro, M. 2005. Heuristic search applied to abstract combat games. In *Conference of the Canadian Society for Computational Studies of Intelligence (Canadian AI 2005)*, volume 3501, 66–78. Springer.

Ontañón, S., and Buro, M. 2015. Adversarial hierarchical-task network planning for complex real-time games. In *IJCAI*, 1652–1658.

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. *TCIAIG* 5(4):293–311.

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

Perkins, L. 2010. Terrain analysis in real-time strategy games: An integrated approach to choke point detection and region decomposition. In *AIIDE*. AAAI Press.

Ponsen, M. J. V.; Gerritsen, G.; and Chaslot, G. 2010. Integrating opponent models with monte-carlo tree search in poker. In *Interactive Decision Theory and Game Theory*.

Rosin, C. D. 2010. Multi-armed bandits with episode context. In *ISAIM*.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; Dieleman, S.; Grewe, D.; Nham, J.; Kalchbrenner, N.; Sutskever, I.; Lillicrap, T.; Leach, M.; Kavukcuoglu, K.; Graepel, T.; and Hassabis, D. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529:484–503.

Stanescu, M.; Barriga, N. A.; and Buro, M. 2014. Hierarchical adversarial search applied to real-time strategy games. In *AIIDE*.

Synnaeve, G., and Bessière, P. 2012. A dataset for StarCraft AI & an example of armies clustering. In *AIIDE*. AAAI Press.

Uriarte, A., and Ontañón, S. 2012. Kiting in RTS games using influence maps. In *AIIDE*. AAAI Press.

Uriarte, A., and Ontañón, S. 2014. Game-tree search over high-level game states in RTS games. In *AIIDE*. AAAI Press.

Uriarte, A., and Ontañón, S. 2015. Automatic learning of combat models for RTS games. In *AIIDE*.

Uriarte, A., and Ontañón, S. 2016. Combat Models for RTS Games.

Weber, B. G., and Mateas, M. 2009. A data mining approach to strategy prediction. In *CIG*. IEEE.