Combinatorial Multi-armed Bandits
for Real-Time Strategy Games

Santiago Ontañón   SANTI@CS.DREXEL.EDU

Computer Science Department, Drexel University
Philadelphia, PA, USA

Abstract

Games with large branching factors pose a significant challenge for game tree search algorithms. In this paper, we address this problem with a sampling strategy for Monte Carlo Tree Search (MCTS) algorithms called *naïve sampling*, based on a variant of the Multi-armed Bandit problem called *Combinatorial Multi-armed Bandits* (CMAB). We analyze the theoretical properties of several variants of *naïve sampling*, and empirically compare it against the other existing strategies in the literature for CMABs. We then evaluate these strategies in the context of real-time strategy (RTS) games, a genre of computer games characterized by their very large branching factors. Our results show that as the branching factor grows, *naïve sampling* outperforms the other sampling strategies.

1. Introduction

Games with large branching factors pose a significant challenge for game tree search algorithms. So far, Monte Carlo Tree Search (MCTS) algorithms (Browne, Powley, Whitehouse, Lucas, Cowling, Rohlfshagen, Tavener, Perez, Samothrakis, & Colton, 2012), such as UCT (Kocsis & Szepesvri, 2006), are the most successful approaches for this problem. The key to the success of MCTS algorithms is that they sample the search space, rather than exploring it systematically. However, MCTS algorithms quickly reach their limit when the branching factor grows. To illustrate this, consider Real-Time Strategy (RTS) games, where each player controls a collection of units, all of which can be controlled simultaneously, leading to a combinatorial branching factor. For example, just 10 units with 5 actions each results in a potential branching factor of $5^{10} \approx 10$ million, beyond what standard MCTS algorithms can handle. Algorithms that can handle adversarial planning in situations with combinatorial branching factors would have many applications to problems such as multiagent planning.

Specifically, this paper focuses on scaling up MCTS algorithms to games with combinatorial branching factors. MCTS algorithms formulate the problem of deciding which parts of the game tree to explore as a Multi-armed Bandit (MAB) problem (Auer, Cesa-Bianchi, & Fischer, 2002). In this paper, we will show that by considering a variant of the MAB problem called the *Combinatorial Multi-armed Bandit* (CMAB) (Gai, Krishnamachari, & Jain, 2010; Chen, Wang, & Yuan, 2013; Ontañón, 2013), it is possible to handle the larger branching factors appearing in RTS games.

Building on our previous work in this area (Ontañón, 2013), where we first introduced the idea of *naïve sampling*, the main contributions of this paper are: (1) an analysis of the different instantiations of the family of *naïve sampling* strategies, including regret bounds; (2) an empirical comparison with other existing CMAB sampling strategies in the literature (LSI, see Shleyfman, Komenda, & Domshlak, 2014; and MLPS, see Gai et al., 2010); (3)

empirical results using increasingly complex situations, to understand the performance of these strategies as the problems grow in size (reaching situations with branching factors in the order of 10^{22}).

We use the μRTS game simulator[1] as our application domain, which is a deterministic and fully-observable RTS game (although it can be configured for partial observability or non-determinism). Our results indicate that for scenarios with small branching factors, *naïve sampling* performs similar to other sampling strategies, but as the branching factor grows, *naïve sampling* starts outperforming the other approaches. A snapshot of all the source code and data necessary to reproduce all the experiments presented in this paper can be downloaded from the author's website[2].

The remainder of this paper is organized as follows. Section 2 presents some background on RTS games and MCTS. Section 3 then introduces the CMAB problem. Section 4 introduces and analyzes naïve sampling strategies, after which Section 5 presents other known sampling strategies for CMABs in the literature. All of these strategies are compared empirically in Section 6. After that, we describe how to integrate them into MCTS in Section 7, and the strength of the resulting MCTS algorithm is evaluated empirically in the μRTS simulator in Section 8. The paper closes with related work, conclusions, and directions for future research.

2. Background

The following two subsections present some background on real-time strategy (RTS) games, and on Monte Carlo Tree Search in the context of RTS games.

2.1 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 (Buro, 2003). Designing AI techniques for RTS games is challenging because:

- They have *huge decision and state spaces*: To have a sense of scale, the worst case branching factor of a typical RTS game, StarCraft, has been estimated to be at least 10^{50} (Ontañón, Synnaeve, Uriarte, Richoux, Churchill, & Preuss, 2013) when the player can control all units simultaneously, which is staggering if we compare it with the branching factors of games like Chess (about 36) and Go (about 180). Moreover, the state space of StarCraft has been estimated to be at least 10^{1685} (Ontañón et al., 2013), compared to about 10^{47}(Chinchalkar, 1996) for Chess and 10^{171}(Tromp & Farnebäck, 2006) for Go.

- 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 move; (2) players do not take turns, but can issue actions simultaneously

---

1. https://github.com/santiontanon/microrts
2. https://sites.google.com/site/santiagoontanonvillar/code/NaiveSampling-journal-2016-source-code.zip

(i.e., two players can issue actions at the same instant of time, and to as many units as they want); and (3) actions are durative, i.e., actions might take more than one decision cycle to complete.

Some RTS games are also partially observable and non-deterministic, but we will not deal with these properties in this paper.

In RTS games, players control a collection of individual units that players issue actions to. Each of these units can only execute one action at a time, but, since there might be multiple units in a game state, players can issue multiple actions at the same time (one per unit they control). We will refer to those actions as *unit-actions*, and use lower case a to denote them. A *player-action* α is the set of unit-actions that one player issues simultaneously at a given time: $\alpha = \{a_1, ..., a_n\}$. Thus, without loss of generality, we can consider that players issue only one *player-action* per game decision cycle (which will consist of as many unit-actions as units ready to execute an action in the current decision cycle). In this way, even if unit-actions are durative, we can see an RTS game as a game where each player issues exactly one player-action at each decision cycle. The number of possible player-actions corresponds to the branching factor. Thus, the branching factor in a RTS game grows exponentially with the number of units each player controls (without loss of generality, we can assume a special *no-op* unit-action, to be issued to those units the player does not want to do anything in the current decision cycle).

To illustrate the size of the branching factor in RTS games, consider the situation from the μRTS game[3] (used in our experiments) 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 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 free adjacent cells. In total, player *max* in Figure 1 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.

Specifically, a two-player deterministic perfect-information RTS game is a tuple $G = (S, A, P, \tau, L, W, s_{init})$, where:

- S is the game state space (e.g., in Chess, the set of all possible board configurations).

- A is the finite set of possible player-actions that can be executed in the game.

- $P = \{max, min\}$ is the set of players.

- $\tau : S \times A \times A \to S$ is the deterministic transition function, that given a state at time t, and the actions of the two players, returns the state at time $t + 1$.

- $L : S \times A \times P \to \{true, false\}$ is a function that given a state, a player-action and a player, determines whether it is legal to execute the given player-action by the given player in the given state. We will write $actions(s, p) = \{\alpha \in A | L(s, \alpha, p) = true\}$ to denote the set of player-actions that player p can execute in state s.

---

3. For gaining a more intuitive idea of μRTS, a gameplay video can be found here: https://www.youtube.com/watch?v=Or3IZaRRYIQ
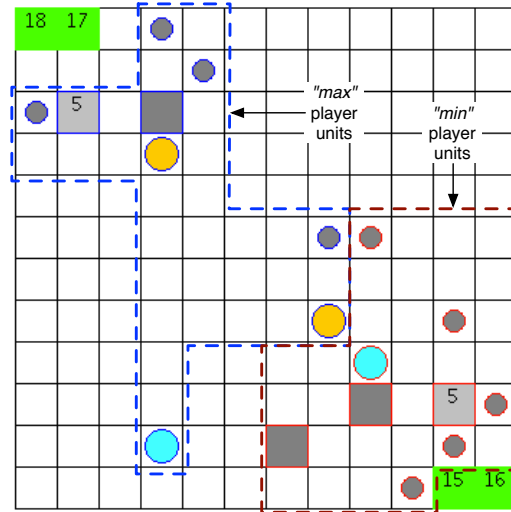
Figure 1: A screenshot of the μRTS simulator. Square units correspond to "bases" (light grey, that can produce workers), "barracks" (dark grey, that can produce military units), and "resources mines" (green, from where workers can extract resources to produce more units), the circular units correspond to workers (small, dark grey) and military units (large, yellow or light blue).

- $W : S \to P \cup \{draw, ongoing\}$ is a function that given a state 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.

In order to apply game tree search, an additional *evaluation function* is typically provided. The evaluation function predicts how attractive is a given state for a player. We will assume an evaluation function of the form $\rho : S \to \mathbb{R}$, which returns positive numbers for states that are good for *max* and negative numbers for states that are good for *min*.

2.2 Monte Carlo Tree Search in RTS Games

Monte Carlo Tree Search (MCTS) is a family of planning algorithms based on sampling the decision space rather than exploring it systematically (Browne et al., 2012). MCTS algorithms maintain a partial game tree. Each node in the tree corresponds to a game state, and the children of that node correspond to the result of one particular player executing actions. Additionally, each node stores the number of times it has been explored, and the average reward obtained when exploring it. Initially, the tree contains a single root node with the initial state. Then, assuming the existence of a reward function ρ, at each iteration of the algorithm the following three processes are executed:

- *SelectAndExpandNode*: Starting from the root node, one of the current node's children is chosen following a *tree policy*, until a node n that was not in the tree before is reached. The new node n is added to the tree.

- *Simulation*: A Monte Carlo *simulation* (a.k.a. a *playout* or a *rollout*) is executed starting from n using a *default policy* (e.g., random) to select actions for all the players until a terminal state or a maximum simulation time is reached. Let $r = \rho(s)$ be the reward in the state s at the end of the simulation.

- *Backup*: r is propagated up the tree, starting from the node n, and continuing through all the ancestors of n in the tree (updating their average reward, and incrementing by one the number of times they have been explored).

When the computation budget is over, the action that leads to the "best" child of the root node of the tree is selected as the best action to perform. Here, "best" can be defined as the one with highest average reward, the most visited one, or some other criteria (depending on the tree policy).

Different MCTS algorithms typically differ just in the tree policy. In particular, UCT (Kocsis & Szepesvri, 2006) frames the tree policy as a *Multi-armed Bandit* (MAB) problem. MAB problems are a class of sequential decision problems, where at each iteration an agent needs to choose amongst k actions (called *arms*), in order to maximize the cumulative reward obtained by those actions. A MAB problem with k arms is defined by a set of unknown real reward distributions $\mu_1, ..., \mu_k$, associated with each of the k arms. Therefore, the agent needs to estimate the potential rewards of each action based on past observations, balancing exploration and exploitation.

UCT uses a specific sampling strategy called UCB1 (Auer et al., 2002) that balances exploration and exploitation of the different nodes in the tree. It can be shown that, when the number of iterations executed by UCT approaches infinity, the probability of selecting a suboptimal action approaches zero (Kocsis & Szepesvri, 2006). However, UCB1 does not scale well to the domains of interest in this paper, where the branching factor might be several orders of magnitude larger than the number of samples we can perform.

Previous work has addressed many of the key challenges arising in applying game tree search to RTS games. For example game tree search algorithms exist that can handle durative actions (Churchill, Saffidine, & Buro, 2012), or simultaneous moves (Kovarsky & Buro, 2005; Saffidine, Finnsson, & Buro, 2012). However, the branching factor in RTS games remains too large for current state-of-the-art techniques.

Many ideas have been explored to improve UCT in domains with large branching factors. For example, *first play urgency* (FPU) (Gelly & Wang, 2006) allows the bandit strategy of UCT (UCB) to exploit nodes early, instead of having to visit all of them before it starts exploiting. However, FPU still does not address the problem of selecting which of the unexplored nodes to explore first (which is key in our domains of interest). Another idea is to try to maximally exploit the information obtained from each simulation, like performed by AMAF (Gelly & Silver, 2007). However, again, this does not solve the problem of having a branching factor many orders of magnitude larger than the number of simulations we can perform. As elaborated in Section 9, three main approaches have been explored to address this problem: (1) use abstraction to represent the game state or the action space to simplify the problem (Balla & Fern, 2009; Uriarte & Ontañón, 2014); (2) portfolio approaches that only consider moves chosen by a predefined portfolio of strategies (Churchill & Buro, 2013; Chung, Buro, & Schaeffer, 2005); or (3) hierarchical search approaches that aim at pruning the search space by considering first high-level decisions, which condition the potential

number of low-level decisions that can be taken (Stanescu, Barriga, & Buro, 2014; Ontañón & Buro, 2015). This paper studies an alternative idea, namely using *combinatorial multi-armed bandits* (CMAB), which have recently been proposed as a solution to address the combinatorial branching factors arising in RTS games (Ontañón, 2013; Shleyfman et al., 2014).

3. Combinatorial Multi-armed Bandits

A *Combinatorial Multi-armed Bandit* (CMAB) is a variation of the MAB problem. We use the formulation in Ontañón (2013), which is more general than that by Gai et al. (2010) or that by Chen et al. (2013). Specifically, a CMAB is defined by:

- A set of n variables $X = \{X_1, ..., X_n\}$, where variable X_i can take K_i different values $\mathcal{X}_i = \{v_i^1, ..., v_i^{K_i}\}$, and each of those values is called an *arm*. Let us call $\mathcal{X} = \{(v_1, ..., v_n) \in \mathcal{X}_1 \times ... \times \mathcal{X}_n\}$ to the set of possible value combinations, where each of these combinations $V \in \mathcal{X}$ is called a *macro-arm*.

- An unknown reward distribution $\mu : \mathcal{X} \to \mathbb{R}$ over each macro-arm.

- A function $L : \mathcal{X} \to \{true, false\}$ that determines which macro-arms are legal.

The problem is to find a legal macro-arm that maximizes the expected reward. Strategies to address CMABs are designed to iteratively sample the space of possible macro-arms. At each iteration t, one macro-arm V_t is selected, which results in a given reward $\mu_t = \mu(V_t)$. Strategies must balance exploration and exploitation in order to converge to the best macro-arm in the shortest number of iterations possible. We will call V_t^{best} to the macro-arm recommended as the best by the given strategy after iteration t.

The difference between a MAB and a CMAB is that in a MAB there is a single variable, whereas in a CMAB, there are n variables. A CMAB can be translated to a MAB, by considering that each possible legal macro-arm is a different arm in the MAB. Moreover, the possible number of macro-arms in a CMAB grows exponentially with the number of variables (depending on how many of those macro-arms are legal).

The performance of strategies to address MABs and CMABs is assessed by measuring the *regret*, which is the difference between the expected reward of the selected macro-arm, and the expected reward of an optimal macro-arm. Moreover, regret can be computed in several different ways (Bubeck, Munos, & Stoltz, 2011), assuming that V^* is an optimal macro-arm, obtaining maximum expected reward $\mu^* = E(\mu(V^*))$:

- *Instantaneous regret*: is the difference between μ^* and the reward obtained by the last selected macro-arm. After T iterations, the instantaneous regret is computed as:

$$r'_T = \mu^* - \mu_T$$

- *Cumulative regret* (referred to as *pseudo-regret* in some texts, see Bubeck & Cesa-Bianchi, 2012): is the sum of differences between μ^* and the reward obtained by the selected macro-arms at each iteration. After T iterations, the cumulative regret is:

$$R_T = \sum_{t=1}^{T} (\mu^* - \mu_t)$$

where μ_t is the reward obtained at iteration t.

- *Simple regret*: after T iterations, the simple regret is the difference between μ^* and the reward obtained by the macro-arm believed to be the best at iteration T:

$$r_T = \mu^* - \mu(V_T^{best})$$

Thus, instantaneous regret is the difference in reward at a given time t based on the selected macro-arm, cumulative regret is the sum of all the instantaneous regrets so far, and simple regret is the instantaneous regret that would be obtained if the arm recommended as the best at iteration t is chosen.

As pointed out by Bubeck et al. (2011), strategies that minimize cumulative regret, obtain larger simple regrets, and vice-versa. Therefore, it is important to determine which kind of regret we must minimize in the task we are modeling using MABs or CMABs. As pointed out by Tolpin and Shimony (2012), and by Shleyfman et al. (2014), bandit strategies applied to planning in games should minimize simple regret, since the performance of the agent in the game is based only on the performance of the final action selected, which corresponds to simple regret. Thus, it might seem that standard approaches to MCTS, such as UCT which uses UCB1 (Auer et al., 2002) and thus minimize cumulative regret, are minimizing the wrong measure. However, notice that bandit strategies running in the search nodes of an MCTS algorithm need to balance two main objectives: (1) identify the best action, and (2) estimate the reward of the best action. While the first one is achieved by doing pure exploration (aiming at minimizing simple regret), the later is not. Thus, bandit strategies in MCTS algorithms need to strike a balance between exploration and exploitation in order to achieve both objectives. Recently, however, several MCTS algorithms have been designed to directly minimize simple regret. Examples are BRUE (Feldman & Domshlak, 2014), MCTS SR+CR (Tolpin & Shimony, 2012), or SHOT (Cazenave, 2015). For simplicity, however, in our experimental evaluation, we will use a standard MCTS algorithm.

In this paper, we will use CMABs to model the decision process that a player faces in RTS games. Each of the units in the game state will be modeled with a variable X_i, and the values that each of these variables can take correspond to the unit-actions that the corresponding units can execute. Player-actions thus naturally correspond to macro-arms.

4. Naïve Sampling For CMABs

Naïve sampling (NS) is a family of sampling strategies based on assuming that the reward distribution μ can be approximated as the sum of a set of reward functions $\mu_1, ..., \mu_n$, each of them depending only on the value of one of the variables of the CMAB:

$$\mu(X) \approx \sum_{i=1...n} \mu_i(X_i)$$

We call this the *naïve assumption*, since it is reminiscent of the conditional independence assumption of the Naïve Bayes classifier. Thanks to the naïve assumption, we can break the CMAB problem into a collection of $n + 1$ MAB problems.

- *Local MABs*: For each $X_i \in X$, we define a MAB, MAB_i, that only considers X_i.

- *Global MAB*: MAB_g, that considers the whole CMAB problem as a MAB where each legal macro-arm that has been sampled so far is one of the candidate arms. This means that in the first iteration, $t = 1$, the global MAB contains no arms at all.

Intuitively, naïve sampling uses the local MABs to *explore* different macro-arms that are likely to result on a high reward, and then uses the global MAB to *exploit* the macro-arms that obtained the highest reward so far. Let us first introduce some notation:

- Let $T_i^t(v_i^k)$ be the number of times that value v_i^k has been selected for variable X_i up to iteration t.

- Let $\overline{\mu}_i^t(v_i^k)$ be the marginalized average reward obtained when selecting value v_i^k for variable X_i up to time t.

- Let $T^t(v_1^{k_1}, ..., v_n^{k_n})$ be the number of times that macro-arm $(v_1^{k_1}, ..., v_n^{k_n})$ has been selected up to time t.

- Let $\overline{\mu}^t(v_1^{k_1}, ..., v_n^{k_n})$ be the average reward obtained when selecting the macro-arm $(v_1^{k_1}, ..., v_n^{k_n})$ up to time t.

The NS strategy works as follows. At each iteration t:

- Use a strategy π_0 to determine whether to *explore* (via the local MABs) or *exploit* (via the global MAB).

  - If *explore* was selected: a legal macro-arm $x^t = (x_1^t, ..., x_n^t)$ is selected by using a strategy π_l to select a value for each $X_i \in X$ independently (i.e., the strategy is used n times, one per variable). x^t is added to the global MAB.
  - If *exploit* was selected: a macro-arm x^t is selected by using a strategy π_g over the macro-arms already present in the global MAB.

Intuitively, when exploring, the naïve assumption is used to select values for each variable, assuming that this can be done independently using the estimated $\overline{\mu}_i^t$ expected rewards. At each iteration, the selected macro-arm is added to the global MAB, MAB_g. Since the assumption is that the number of iterations T that we can perform is much smaller than N (the total number of macro-arms), it is expected that almost each time that the strategy decides to explore, the selected macro-arm is not going to be already in the global MAB.

When exploiting, MAB_g is used to sample amongst the explored macro-arms, and find the one with the expected maximum reward. Thus, we can see that the naïve assumption is used to explore the combinatorial space of possible macro-arms, and then a regular MAB strategy is used over the global MAB to select the optimal macro-arm. If the strategy π_l is selected such that each arm has a non-zero probability of being selected, then each possible value combination also has a non-zero probability. Thus, the error in the estimation of $\overline{\mu}^t$ constantly decreases. As a consequence, the optimal value combination will eventually have the highest estimated reward, regardless of whether the reward function violates the naïve assumption or not. Thus, notice that in order to work well, naïve sampling only requires

the game domain to satisfy the naïve assumption loosely (i.e., that macro-arms composed of unit-actions that have individually high reward, also tend to have a high reward).

Moreover, NS is not just one sampling strategy, but a whole family of sampling strategies, since we still need to decide which sampling strategies to use for π_0, π_l, and π_g. In our previous work, we studied the performance when all three strategies are ϵ-greedy strategies (Ontañón, 2013). Let us now analyze the behavior of NS for different instantiations of these strategies. The following subsections first present the theoretical regret bounds, and then an empirical comparison of the performance of these strategies in the μRTS simulator. Evaluation of the performance of these strategies in the context of game tree search is presented in Section 8.

4.1 ϵ-greedy Naïve Sampling

One of the most common MAB sampling strategies is ϵ-greedy. An ϵ-greedy strategy with parameter $0 \leq \epsilon \leq 1$ selects the arm considered to be the best one so far with probability $1 - \epsilon$, and with probability ϵ it selects one arm at random. If ϵ is small (e.g., 0.1), this results on a behavior that selects the arm currently considered the best most of the times, but keeps exploring all the other arms with a small probability.

We will write NS(ϵ_0,ϵ_l,ϵ_g) to denote a naïve sampling where π_0, π_l, and π_g are ϵ-greedy strategies, with parameters $0 \leq \epsilon_0 \leq 1$ (ϵ_0 probability of selecting *explore* and $1 - \epsilon_0$ of selecting *exploit*), $0 \leq \epsilon_l \leq 1$, and $0 \leq \epsilon_g \leq 1$ respectively. Let us now see how the regret of this strategy grows over time.

Proposition 1. *The* cumulative regret *of NS(ϵ_0,ϵ_l,ϵ_g) grows linearly as $R_T = O((1 - p^*)DT)$, where T is the number of iterations, D is the expected difference in expected reward between an optimal macro-arm and a non-optimal macro-arm, and $p^* \geq (1 - \epsilon_0)(1 - \epsilon_g)$ is the probability of selecting an optimal arm when $T \to \infty$ (proof in Appendix A).*

Moreover, if we assume a single optimal arm and we know that the reward function μ satisfies the naïve assumption, we can be more precise, and get an exact value for p^*:

$$p^* = (1 - \epsilon_0)\left[(1 - \epsilon_g) + \frac{\epsilon_g}{N}\right] + \epsilon_0 \prod_{i=1...n}\left[(1 - \epsilon_l) + \frac{\epsilon_l}{K_i}\right]$$

Where N is the total number of legal macro-arms.

Proposition 2. *The* simple regret *of NS(ϵ_0,ϵ_l,ϵ_g) decreases at an exponential rate as $r_T = O(De^{-2d^2Tp_i})$, where $p_i \geq \epsilon_0\Pi_{j=1...n}\frac{\epsilon_l}{K_j} + (1 - \epsilon_0)\frac{\epsilon_g}{N}$, D is as in Proposition 1, and d is the minimum difference in reward between an optimal macro-arm and a non-optimal macro-arm (proof in Appendix A).*

So, in summary, naïve sampling has linear cumulative regret (which means that even after a very long number of iterations, it will still pick suboptimal arms with a fixed probability) and exponentially decreasing simple regret (which means that the probability of the arm believed to be the best at the very end of the execution not to be optimal decreases exponentially with the number of iterations executed). This is expected, since, it inherits these properties from ϵ-greedy, which also has linear cumulative regret and exponentially

decreasing simple regret (see Appendix A). Also notice that as presented here, ϵ-greedy naïve sampling is a strict generalization of ϵ-greedy. If $\epsilon_l = 1.0$ and $\epsilon_g = 0.0$, ϵ-greedy naïve sampling is equivalent to an ϵ-greedy policy with parameter ϵ_0. Also, although variations of the ϵ-greedy strategy are known that have logarithmic cumulative regret (e.g., see Auer et al., 2002), we will not explore those in this paper.

Moreover, notice the interesting inverse relation between simple regret and cumulative regret (already noted by Bubeck et al., 2011). According to the previous propositions, to minimize cumulative regret, we need to make ϵ_g and ϵ_l as small as possible, and to minimize simple regret, we need to make ϵ_g and ϵ_l as large as possible. So, if we were mostly interested in simple regret in the context of RTS games, this points out that larger values might result in stronger game play. This is echoed in our experimental results, where the best performance was achieved with relatively high values for ϵ_l (0.4). Notice, moreover, that setting ϵ_l and ϵ_g to 1 (as might seem to be suggested by the results of the propositions), would not work well in practice. The reason is that the proposition results concern a large computational budget (larger than the number of arms). For a smaller computational budget (more realistic in practice), large values of ϵ_l and ϵ_g will just result in never sampling any arm more than once, leading to a very poor estimate of their rewards, and thus to low performance. For that reason, our experiments indicate that relatively large (but not all the way to 1) values of ϵ_l achieve the best results.

Thus, the upper bounds in the previous propositions are for a sufficiently large number of iterations. However, the key problem in CMABs is that we assume that the number of iterations we can perform is small compared to the number of possible macro-arms. Therefore, it is interesting to analyze the behavior of these strategies when the number of iterations is small. For that purpose, Section 6 presents an empirical comparison of the different strategies presented in this paper.

4.2 Two-Phase Naïve Sampling

Under the assumption that number of iterations we can perform is much smaller than the total number of macro-arms, $T \ll N$, the global MAB will never reach the point of having all possible macro-arms. Moreover, as pointed out by Shleyfman et al. (2014), it does not make sense to consider new macro-arms toward the end of the computation budget, since we are not going to have enough time to obtain accurate estimations of their expected reward. This motivates sampling strategies which vary their exploration and exploitation trade-offs over time (e.g., starting with a large probability of exploration, and gradually reducing it). An example of such strategy for regular MABs is the *decreasing ϵ-greedy* sampling strategy (Auer et al., 2002). In this paper, we will focus on the simplest instantiation of these strategies: *two-phase* sampling strategies, which instead of gradually changing the probability of exploration, they perform a first "exploration" phase to find a set of candidate macro-arms, and in a second phase they try to find the best macro-arm, only amongst those explored during the first phase. An example two-phase strategy for CMABs is LSI (Shleyfman et al., 2014) (described in Section 5.2).

We will write $NS(k, \epsilon_0^1, \epsilon_l^1, \epsilon_g^1, \epsilon_0^2, \epsilon_l^2, \epsilon_g^2)$ to denote a naïve sampling strategy where parameters $\epsilon_0^1, \epsilon_l^1, \epsilon_g^1$ are used during the first k sampling iterations, and $\epsilon_0^2, \epsilon_l^2, \epsilon_g^2$ are used during the rest of the iterations. Following the intuition above, the parameters in the first phase

should be geared toward exploring (high values for all the parameters) and the ones in the second phase, should be geared toward exploiting (low values for all the parameters).

If the computation budget T is known ahead of time, k can be set to a fraction r of the total computation budget ($k = rT$). In that case, we will denote the strategy by $NS(rT, \epsilon_0^1, \epsilon_l^1, \epsilon_g^1, \epsilon_0^2, \epsilon_l^2, \epsilon_g^2)$.

The theoretical analysis of the two-phase naïve sampling strategy is very similar to the one-phase case, with the exception of one interesting case (when the first phase lasts for a finite number of iterations k, and $\epsilon_0^2 = 0$), which is the only case we will consider here (in all other cases, cumulative regret grows linearly and simple regret decreases exponentially as in the one-phase case). Let us start by bounding the probability of sampling the optimal macro-arm at least once during the first phase.

Proposition 3. *In a CMAB with n variables, the probability that after t iterations using a $NS(\epsilon_0, \epsilon_l, \epsilon_g)$ sampling strategy an optimal macro-arm V^* has not been explored at least once, decreases exponentially as a function of t, and is at most $(1-p)^{t\epsilon_0}$, where $p = \prod_{i=1...n} (\epsilon_l/K_i)$ (proof in Appendix A).*

This means that in order to maximize the probability of having V^* among the explored macro-arms during the first phase, we want to maximize both ϵ_0^1 and ϵ_l^1. Given Proposition 3, we can now analyze the behavior when k is a finite number of rounds, and $\epsilon_0^2 = 0$.

Proposition 4. *The cumulative regret of $NS(k, \epsilon_0^1, \epsilon_l^1, \epsilon_g^1, \epsilon_0^2, 0, \epsilon_g^2)$ when k is a constant grows linearly when $T \gg k$:*

$$R_T = O\left(t\left[(1 - \epsilon_g - q_k + \epsilon_g q_k)d + \epsilon_g D\right]\right)$$

where $q_k = 1 - (1 - p)^{k\epsilon_0^1}$, and $p = \prod_{i=1...n} \left(\epsilon_l^1/K_i\right)$. (proof in Appendix A).

Notice this is worse than the one-phase case in the limit, since it grows faster.

Proposition 5. *The simple regret of $NS(k, \epsilon_0^1, \epsilon_l^1, \epsilon_g^1, \epsilon_0^2, 0, \epsilon_g^2)$ when k is a constant and $T \gg k$ is lower bounded by $(1 - q_k)d$, where $q_k = 1 - (1 - p)^{k\epsilon_0^1}$, and d is difference between the best non-optimal macro-arm and an optimal macro-arm. (proof in Appendix A).*

This means that in the particular case where $\epsilon_l^2 = 0$ and k is a constant, even after a very large number of iterations, the probability of selecting a suboptimal arm at the very end will not approach zero, since it will depend on wether the optimal macro-arm was explored during the first phase or not.

Moreover, notice that even if in theory, the asymptotic regret bounds seem to be worse than the one-phase strategy, in practice, a two-phase strategy might work better in some scenarios, since what matters in practice (and in the specific application domain of RTS games) is their behavior for small computational budgets (this is evaluated in Section 6.2).

4.3 Naïve Sampling Beyond ϵ-greedy

An interesting question is whether MAB sampling strategies such as UCB1 (Auer et al., 2002), commonly used in MCTS algorithms can improve over naïve sampling using ϵ-greedy. One problem of UCB1 is that it requires exploring each arm at least once (unless strategies

like *First Play Urgency*, FPU, are used, see Gelly & Wang, 2006), which is problematic, since the number of macro-arms is very large.

Although we do not provide any theoretical results for this strategy, in the experiments below, we experimented with using UCB1 as the sampling strategy for the global MAB in regular naïve sampling. We will write $NS(\epsilon_0, \epsilon_l, UCB1)$ to denote this strategy.

5. Other CMAB Sampling Strategies

Two other sampling strategies for CMABs exist in the literature: MLPS (Gai et al., 2010) and LSI (Shleyfman et al., 2014), which we summarize here. A third algorithm CUCB (Chen et al., 2013) exists, but it is restricted to the specific case where all the variables are boolean (choosing a macro-arm corresponds to choosing a subset of the variables in the CMAB), and thus, we do not include it in our analysis.

5.1 Matching Learning with Polynomial Storage (MLPS)

MLPS (Matching Learning with Polynomial Storage) was presented by Gai et al. (2010) for the problem of multiuser channel allocation. MLPS works in a very similar way to the exploration part of naïve sampling. Specifically, MLPS keeps the same $T_i^t(v_i^k)$ and $\overline{\mu}_i^t(v_i^k)$ estimates for the values that each variable can take. The main difference with respect to naïve sampling is in the way the macro-arm is selected at each iteration. MLPS assumes that all the variables can take the same values (i.e., $\forall i, j : \mathcal{X}_i = \mathcal{X}_j$), and thus uses the Hungarian algorithm (Kuhn, 1955) to find the macro-arm that maximizes the expression:

$$W_V^t(n) = \sum_{v_i^k \in V} \overline{\mu}_i^t(v_i^k) + CM\sqrt{\frac{(M+1)\ln t}{min_{v_i^k \in V} T_i^t(v_i^k)}}$$

Where M is the number of values a variable can take, and C is the exploration parameter (in the original paper, Gai et al. set $C = 1$). In our more general CMAB setting, where each variable has an arbitrary number of values, and where we have an additional function L that determines which macro-arms are legal, the Hungarian algorithm cannot be used directly. Thus, in the experiments presented below, we replaced the Hungarian algorithm with a greedy approach as follows:

1. Start with an empty macro-arm V.

2. Select a random $X_i \in X$ that does not yet have a value.

3. Select the value for X_i which, given the previous selected values, maximizes $W_V^t(n)$.

4. Repeat until all variables have a value.

The previous process is repeated a certain number of times (10 in our experiments), and the iteration that resulted in the highest value is selected. Moreover, we set M in the equation above to be the number of values of the variable that has the most possible values. Although this does not ensure selecting the macro-arm that maximizes $W_V^t(n)$, it is an efficient approach, suitable for real-time games. To distinguish this adapted MLPS strategy from the original MLPS strategy, we will refer to it as $MLPS_{greedy}$.

5.2 Linear Side Information (LSI)

LSI (Shleyfman et al., 2014) is a family of two-phase sampling strategies based on the following idea: while naïve sampling interleaves exploration and exploitation, LSI splits the computation budget $T = T_g + T_e$ into a first *candidate generation* phase (with computation budget T_g) and a second *candidate evaluation* phase (with computation budget T_e). During candidate generation, LSI first collects *side information* (analogous to the $\bar{\mu}_i^t(v_i^k)$ estimates in naïve sampling), and then, using that information, it generates k candidate macro-arms. During the second phase, LSI uses *sequential-halving* (Karnin, Koren, & Somekh, 2013) to determine the best of the k macro-arms. During candidate generation, the computation budget T_g is divided equally among all the different values of the different variables in the CMAB. The different LSI strategies differ in the way these T_g samples are used to collect the side information, and how this side information is used to generate the candidates:

- LSI$_V$: assuming that there is a value for each variable that is special (in the case of RTS games, the "no action" unit-action), the computation budget of each action is used by setting the value of all the other variables to this special value. In this way, LSI obtains an estimate of how much each value of each variable contributes to the global reward (assuming a linear contribution).

- LSI$_F$: the computation budget for each value is used by setting random values for all the other variables.

Once the candidate generation computation budget is spent, LSI estimates the expected contribution of each value of each variable to the overall reward. Using this estimation, two strategies for generating candidate macro-arms are proposed:

- LSI^e (entropy): first, the variables of the CMAB are sorted in decreasing order of entropy, where entropy of a variable is calculated as the entropy of the set of estimated rewards for each of the values of the variable. Intuitively a variable with high entropy is one where the expected reward of its different possible values are very different to each other, while in a variable with low entropy, the expected reward for all of its values will be very similar. Using this order, the variables are then sampled one by one to generate new macro-arms. To sample a value for each variable, the vector of expected rewards for each value of the variable is normalized, so it forms a probability distribution, which is used to generate a value for this variable. Notice that sorting the variables is useful since selecting a value for a variable might prevent selecting certain values for other variables. Thus, sampling first those variables that have a high entropy ensures that the variables that have a larger impact on the expected reward are sampled without having any of their values forbidden by some prior choice.

- LSI^u (union): instead of sorting the variables, the union of all the values of all the variables that still do not have a value is used for sampling the next value to add to the macro-arm, until the macro-arm is complete (i.e., it has a value for each variable).

Once k candidate macro-arms have been generated, sequential-halving is used to determine the best one, with the remaining computational budget. In our experiments, we used the LSI$_V^e$, which has been reported to obtain the best results (Shleyfman et al., 2014). For a more detailed description of LSI, the reader is referred to the work of Shleyfman et al. (2014).

Figure 2: μRTS situations corresponding to CMAB$_1$ (left) and CMAB$_3$ (right) used in the experiments presented in this paper. Exact situation definitions can be found in the source code link provided in the introduction.

6. Empirical Comparison of CMAB Sampling Strategies

In order to illustrate the performance of naïve sampling compared to other sampling strategies for MABs or CMABs this section presents an empirical comparison. For this comparison, we employed three CMABs (with an increasing number of macro-arms), corresponding to three specific situations in μRTS:

- CMAB$_1$: used by Shleyfman et al. (2014) corresponds to the situation shown on the left-hand side of Figure 2, from the perspective of the blue player (max). It has 12 variables, and a total of 10,368 legal macro-arms.

- CMAB$_2$: which corresponds to the situation depicted in Figure 1 from the perspective of the blue player (max). CMAB$_2$ has 9 variables (corresponding to the 9 units controlled by the blue player) and a total of 1,008,288 legal macro-arms.

- CMAB$_3$: corresponds to a larger situation (right-hand side of Figure 2, from the perspective of the blue player, max). This is a 16×16 map, with 110 variables (although only 50 can take more than 1 value) and 9.28×10^{22} legal macro-arms[4].

Notice that these numbers of macro-arms are many orders of magnitude larger than the number of arms typically considered in MABs. We evaluate the following strategies:

- ϵ-**greedy** (treating the CMABs as if they were actually MABs): we show results for $\epsilon = 0.25$ and $\epsilon = 0.5$ (we tested values between 0.0 and 1.0 at intervals of 0.125 and show the ones that obtained the best results). V_t^{best} is selected based on which arm has been sampled most often at iteration t.

---

4. The number of macro-arms was calculated with the built-in branching factor calculator of μRTS.

- **UCB1** (Auer et al., 2002) (also considering MABs): given that the number of macro-arms is larger than the number of samples we can perform, the value of the exploration parameter C of UCB1 has no effect in this experiment (in fact, using UCB1 when there are more arms than the number of iterations that can be run is hopeless, but we included it in our analysis just to set a baseline). However, we set it to $C = 0.05$, which achieved the best results when we combine UCB1 with FPU (below). V_t^{best} is selected based on which arm has been sampled most often at iteration t (ties are resolved by selecting the arm with the highest expected evaluation so far).

- **UCB1-FPU** (Gelly & Wang, 2006) (also considering MABs): we set the FPU constant to 0.51, 0.56 and 0.60 for each of the three CMABs used in our evaluation. These values achieved the best results in our evaluation (in a deployed system we would not be able to change this value depending on the situation, but we wanted to show the best that UCB1-FPU can achieve in each scenario), and $C = 0.05$. V_t^{best} is selected based on which arm has been sampled most often at iteration t.

- **MLPS**$_{greedy}$ (a variation of MLPS, see Gai et al., 2010, as described above): we used $C = 0.005$ for this strategy, which achieved the best results in our experiments.

- **LSI**$_V^e$ (Shleyfman et al., 2014): the linear side information strategy described above. We divided the computation budget as $T_g = 0.25 \times T$ and $T_e = 0.75 \times T$, which achieved the best results in our experiments.

- **NS**(ϵ_0,ϵ_l,ϵ_g): ϵ-greedy naïve sampling strategy. We used $\epsilon_0 = 0.8$ (emphasizing exploration), $\epsilon_l = 0.4$, and $\epsilon_g = 0$ (emphasizing the fact that the global MAB is used for exploitation), since they achieved the best results in our experiments. V_t^{best} is selected based on which arm has been sampled most often at iteration t.

In order to compare the strategies, we evaluate the expected reward of the arm that would be selected as the best at each iteration (V_t^{best}), i.e., the simple regret. As the reward function, we use the result of running a Monte Carlo simulation of the game during 100 game cycles (using a random action selection strategy), and then using an evaluation function to the resulting game state. As the evaluation function, we used one of the built-in function in μRTS (*SimpleSqrtEvaluationFunction3*). Given a state s, this evaluation function (inspired by the standard LTD2 function, see Churchill et al., 2012) assigns a *score* to each player (*max* and *min*) by summing the resource cost of each of her units, weighted by the square root of their health. Then, it produces a normalized evaluation (in the interval $[-1, 1]$) as: $E(s) = \frac{2*score(max)}{score(min)+score(max)} - 1$. This is the evaluation function used in all the experiments reported in this paper.

6.1 Experiment 1: Comparison of CMAB Strategies in CMABs of Increasing Complexity

Figure 3 shows the average expected reward for a collection of sampling strategies in all three CMABs when sampling between 100 and up to 10,000 iterations (reward ranges from -1 to 1). To measure the performance of the best action generated at each point in time, we compute the average of 200 Monte Carlo simulations of the game during 100 cycles,
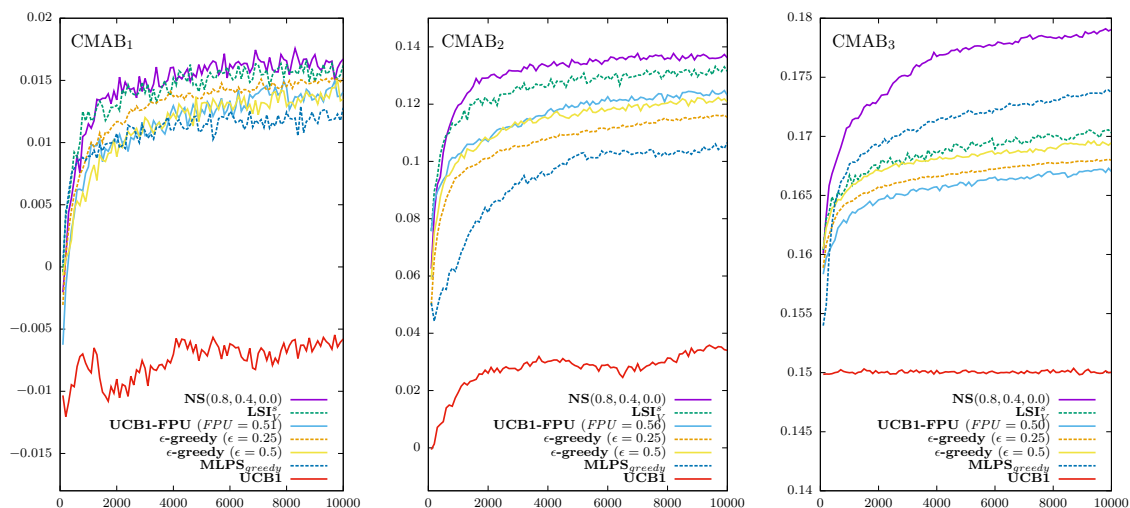
Figure 3: Evolution of the expected reward (vertical axis) of the best arm (V_t^{best}) as the computational budget grows (horizontal axis) on three different CMABs. From left to right: CMAB$_1$ has 10,368 legal macro-arms, CMAB$_2$ has 1,008,288 legal macro-arms and CMAB$_3$ has about 9.28×10^{22}.

Table 1: 95% confidence intervals of the results reported in Figure 3 after 10,000 iterations. We used bold text to highlight the strategies whose intervals overlap with the one achieving the highest reward.

	CMAB$_1$	CMAB$_2$	CMAB$_3$
NS$(0.8, 0.4, 0.0)$	**0.0151 - 0.0184**	**0.1345 - 0.1379**	**0.1785 - 0.1796**
LSI$_V^e$	**0.0146 - 0.0178**	0.1295 - 0.1334	0.1698 - 0.1708
UCB1-FPU	**0.0127 - 0.0155**	0.1212 - 0.1250	0.1665 - 0.1676
ϵ-greedy ($\epsilon = 0.25$)	**0.0137 - 0.0166**	0.1127 - 0.1181	0.1665 - 0.1695
ϵ-greedy ($\epsilon = 0.5$)	0.0122 - 0.0148	0.1185 - 0.1230	0.1690 - 0.1699
MLPS$_{greedy}$	0.0111 - 0.0144	0.1039 - 0.1093	0.1731 - 0.1742
UCB1	-0.0064 - -0.0052	0.0278 - 0.0403	0.1490 - 0.1511

and then apply the same evaluation function described above. The plots are the average of repeating the experiment 100 times. As can be seen, naïve sampling clearly outperforms the other strategies, since the bias introduced by the naïve assumption helps in quickly selecting good player-actions. UCB1 basically did a random selection, since it requires exploring each action at least once, and there are more than 10,000 legal macro-arms in all CMABs. UCB1-FPU performed much better, but still significantly below naïve sampling. In fact, UCB1-FPU performs similar to a simple ϵ-greedy in CMAB$_1$, better in CMAB$_2$ and significantly worse in CMAB$_3$. MLPS$_{greedy}$ only performed competitively in the larger CMAB$_3$, but still far from naïve sampling. LSI$_V^e$ is the strategy that gets closest to naïve sampling, being very close in CMAB$_1$; however, in CMAB$_3$, where the number of macro-

arms is significantly larger, the advantage of naïve sampling is obvious. In order to assess the statistical significance of the results, Table 1 reports the 95% confidence intervals of the average expected reward reported in Figure 3 after sampling 10,000 iterations. When the 95% confidence intervals of two strategies do not overlap, we can say that their difference is statistically significant ($p = 0.05$). As we can see, the difference in reward between naïve sampling and the other strategies is statistically significant in CMAB$_2$ and CMAB$_3$, but not in CMAB$_1$. Specifically, these difference become statistically significant in CMAB$_2$ after 1000 iterations, and in CMAB$_3$ after only 300 iterations. This highlights the advantage of naïve sampling in larger CMABs.

It is interesting to note that since this evaluation is closely related to *simple regret*, rather than *cumulative regret*, strategies that perform more exploration tend to work better. That is why, for example, the best performance for ϵ-greedy in the larger CMAB$_2$ and CMAB$_3$ was achieved with a relatively high $\epsilon = 0.5$. Also, this evaluation seems to contradict results reported by Shleyfman et al. (2014), however, notice that Shleyfman et al. used a version of naïve sampling that determined the best arm as the one with the highest expected reward so far, rather than selecting the most sampled one so far (as we do here).

Finally, notice that the advantage of naïve sampling seems to increase in larger CMABs (e.g., CMAB$_3$). This is because naïve sampling exploits the structure in the domain, and if a value for a given variable is found to obtain a high reward in average, then other macro-arms that contain such value are likely to be sampled. Thus, it exploits the fact that macro-arms with similar values might have similar expected rewards. MLPS also exploits this fact, but since it does not keep a separate *global MAB*, as naïve sampling does, it cannot pinpoint which was the exact combination of values that achieved the highest expected reward.

We would like to note that there are existing strategies, such as HOO (Bubeck, Munos, Stoltz, & Szepesvari, 2008), designed for continuous actions, that can exploit the structure of the action space, as long as it can be formulated as a topological space. Attempting such formulation, and comparing with HOO is part of our future work.

6.2 Experiment 2: Variations of Naïve Sampling

In this section we compare a variety of naïve sampling configurations using the same methodology used in the previous subsection. We used the following configurations (in all of them V_t^{best} is selected based on which arm has been sampled most often at iteration t):

- **NS(ϵ_0,ϵ_l,ϵ_g)**: we employed the same values as before ($\epsilon_0 = 0.8$, $\epsilon_l = 0.4$, and $\epsilon_g = 0.0$).

- **NS($rT, \epsilon_0^1, \epsilon_l^1, \epsilon_g^1, \epsilon_0^2, \epsilon_l^2, \epsilon_g^2$)**: after exploring the space of value combinations at intervals of 0.1, the values that performed better are:

  - $r = 0.6$, $\epsilon_0^1 = 0.8$, $\epsilon_l^1 = 0.4$, $\epsilon_g^1 = 0.0$, and $\epsilon_0^2 = 0.0$, $\epsilon_l^2 = 0.0$, $\epsilon_g^2 = 0.0$. This means doing standard naïve sampling for 60% of the computation budget, and then do pure exploitation during the remaining 40%.

  - $r = 0.6$, $\epsilon_0^1 = 0.8$, $\epsilon_l^1 = 0.4$, $\epsilon_g^1 = 0.0$, and $\epsilon_0^2 = 0.0$, $\epsilon_l^2 = 0.0$, $\epsilon_g^2 = 0.2$. This means doing standard naïve sampling for 60% of the computation budget, and then do ϵ-greedy with $\epsilon = 0.2$ over the set of macro-arms explored during the first phase.
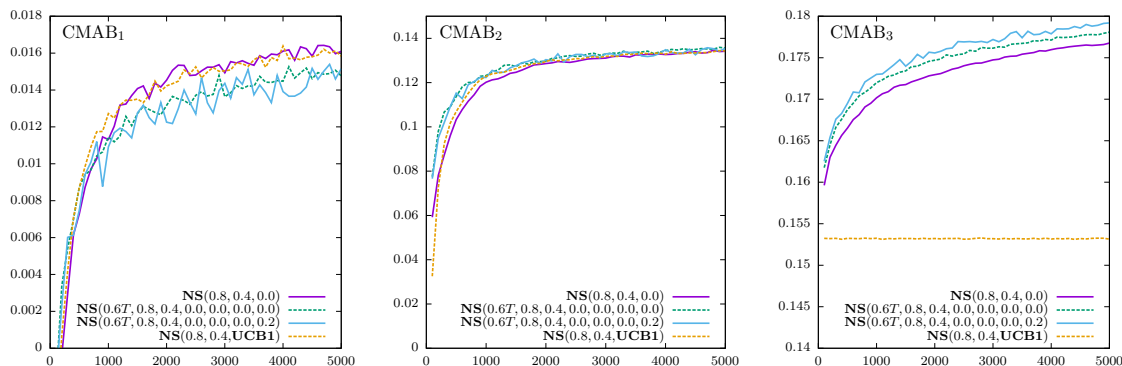
Figure 4: Evolution of the expected reward (vertical axis) of the best arm (V_t^{best}) as the computational budget grows (horizontal axis) on the same three CMABs as Figure 3, for three variants of naïve sampling.

Table 2: 95% confidence intervals of the results reported in Figure 4 after 5,000 iterations. Statistically significant differences are only observed in CMAB_3, highlighted in bold.

	CMAB_1	CMAB_2	CMAB_3
NS$(0.8, 0.4, 0.0)$	0.0149 - 0.0173	0.1326 - 0.1357	0.1760 - 0.1775
NS$(0.6T, 0.8, 0.4, 0.0, 0.0, 0.0, 0.0)$	0.0136 - 0.0159	0.1338 - 0.1374	0.1777 - 0,1785
NS$(0.6T, 0.8, 0.4, 0.0, 0.0, 0.0, 0.2)$	0.0140 - 0.0163	0.1337 - 0.1373	**0.1788 - 0.1796**
NS$(0.8, 0.4, \text{UCB1})$	0.0147 - 0.0170	0.1324 - 0.1363	0.1520 - 0.1543

- **NS**$(\epsilon_0, \epsilon_l, \textbf{UCB1})$: we used $\epsilon_0 = 0.8$, $\epsilon_l = 0.4$, and UCB1 (with $C = 0.005$) as the strategy to sample the global MAB.

Results are shown in Figure 4 in all three CMABs when sampling between 100 and up to 5,000 iterations, evaluated as in the previous experiment (reward also ranges from -1 to 1). Results show that the relative performance of naïve sampling variants depends on the specific CMAB. For example, in CMAB_1, with a "small" branching factor (10,368), a two-phase approach seems to perform worse than standard naïve sampling, or UCB-based naïve sampling (which perform almost identically). In CMAB_2, which has a larger branching factor, the two-phase approach starts to pay off, and performs just slightly better than the other two approaches. Finally, in CMAB_3, with a very large branching factor (about 9.28×10^{22}), the two-phase approach clearly outperforms the other two variants (with the **NS**$(0.6T, 0.8, 0.4, 0.0, 0.0, 0.0, 0.2)$ outperforming all other approaches). Moreover, it is interesting to note that in CMAB_3, using a UCB1 sampling strategy for the global MAB does not work, since the branching factor is so large, that the local MABs never select the same macro-arm twice, and thus, the number of arms in the global MAB is larger than the number of times UCB1 is called. However, notice that for very small computation budgets (smaller than 1000), the two-phase approach outperforms standard naïve sampling in all CMABs (we will get back to this point later in Section 8.4). Table 2 reports the

95% confidence intervals of the values obtained by these different strategies after 5,000 iterations, showing that differences are only statistically significant in $CMAB_3$, where a two phase strategy dominates all the others. In fact in $CMAB_3$, the difference in reward of both two phase strategies with respect to standard naïve sampling is statistically significant as early as after 100 iterations. In $CMAB_2$ the different is initially statistically significant, but it stops being so after 600 iterations.

The conclusion is that for very large branching factors, a two-phase approach pays off, since the last iterations are spent just trying to narrow down, from the set of macro-arms already sampled, which are the best. When the number of macro-arms is not that large, this does not appear to yield any benefit with respect to standard naïve sampling.

7. Monte Carlo Search based on CMABs for RTS Games

As mentioned in Section 3, although many authors have argued for the need of Monte Carlo Tree Search algorithms that use MAB strategies that minimize simple regret instead of cumulative regret, in this paper we will use two standard Monte Carlo search algorithms to evaluate the different CMAB strategies in the context of RTS games:

- a Monte Carlo Tree Search (MCTS) approach (since RTS games involve simultaneous and durative actions, we used the MCTS approach described in our previous work, see Ontañón, 2013), described below, and

- a plain Monte Carlo (MC) search approach (which was implemented basically by limiting the depth of the tree in MCTS to 1).

The main difference between MC and MCTS in our context is that the MC approach does not construct a game tree, as the MCTS approach does. Moreover, some strategies, such as LSI, require knowing the sampling budget before-hand, and thus cannot be used in the context of MCTS (since we cannot anticipate the budget for any tree node except for the root). We used NaïveMCTS (Ontañón, 2013) as our MCTS approach, specifically designed for RTS games.

The first consideration that NaïveMCTS does is that unit-actions in an RTS game are durative (they might take several game cycles to complete). For example, in μRTS, a *worker* takes 10 cycles to move one square in any of the 4 directions, and 200 cycles to build a *barracks*. This means that if a player issues a move action to a *worker*, no action can be issued to that *worker* for another 10 cycles. Thus, there might be cycles in which one or both players cannot issue any actions, since all the units are busy executing previously issued actions. The game tree generated by NaïveMCTS takes this into account, using the same idea as the ABCD (α-β Considering Durations) algorithm (Churchill et al., 2012).

NaïveMCTS is designed for deterministic two-player zero sum games, where one player, *max*, attempts to maximize the evaluation function ρ, and the other player, *min*, attempts to minimize it. NaïveMCTS differs from other MCTS algorithms in the way nodes are selected and expanded in the tree (the *SelectAndExpandNode* procedure in Section 2.2).

The *SelectAndExpandNode* process for NaïveMCTS is shown in Algorithm 1. The process receives a game tree node n_0 as the input parameter, and lines 1-5 determine whether this node n_0 is a *min* or a *max* node (i.e. whether the children of this node correspond

---

Algorithm 1 SelectAndExpandNode(n_0)

1: **if** canMove($max, n_0.state$) **then**
2:    $player = max$
3: **else**
4:    $player = min$
5: **end if**
6: α = NaïveSampling($n_0.state, player$)
7: **if** $\alpha \in n_0.children$ **then**
8:    **return** NaïveSelectAndExpandNode($n_0.child(\alpha)$)
9: **else**
10:    n_1 = newTreeNode(fastForward($n_0.state, \alpha$))
11:    n_0.addChild(n_1, α)
12:    **return** n_1
13: **end if**

---

to moves of player min or of player max). Then, line 6 uses naïve sampling to select one of the possible player-actions of the selected player in the current state. If the selected player-action corresponds to a node already in the tree (line 8), then *SelectAndExpandNode* is recursively applied from that node (i.e. the algorithm goes down the tree). Otherwise (lines 10-12), a new node is created by executing the effect of player-action α in the current game state using the *fastForward* function. *fastForward* simulates the evolution of the game until reaching a decision point (when any of the two players can issue an action, or until a terminal state has been reached). This new node is then returned as the node from where to perform the next simulation.

A final consideration is that RTS games are simultaneous-action domains, where more than one player can issue actions at the same instant of time. Algorithms like minimax might result in under or overestimating the value of positions, and several solutions have been proposed (Kovarsky & Buro, 2005; Saffidine et al., 2012). However, we noticed that this had a very small effect on the practical performance of our algorithm in RTS games, so we have not incorporated any of these techniques into NaïveMCTS.

8. Experimental Results in the Context of Game Tree Search

The following subsections present three separate experiments aimed at evaluating different CMAB sampling strategies in the context of Game Tree search.

8.1 Experimental Setup

In order to evaluate the performance of the different CMAB sampling strategies, as before, we employed μRTS. μRTS games used in this paper are fully observable and deterministic, but still capture several defining features of full-fledged RTS video games: durative and simultaneous actions, large branching factors, resource allocation, and real-time combat.

We employed eight different μRTS maps, that result in games of different average branching factors (Figure 5):
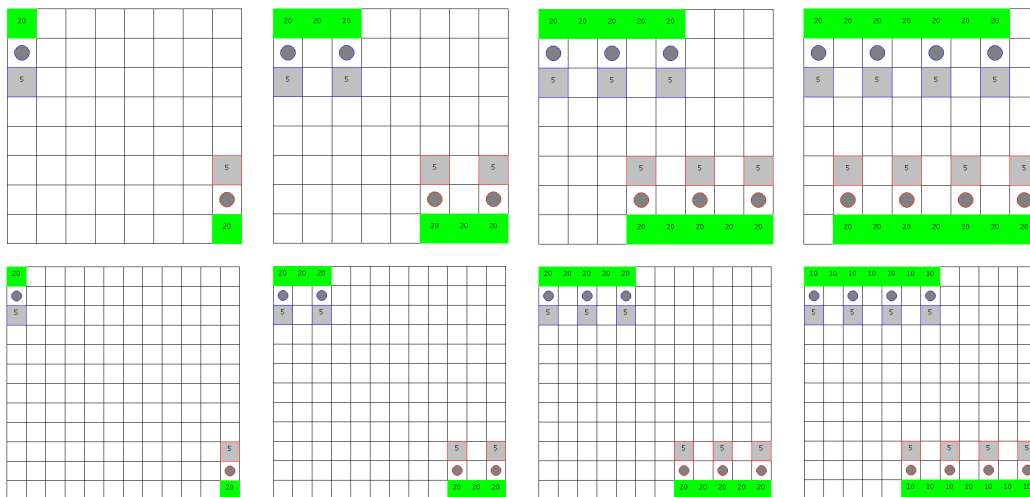
Figure 5: The eight maps used in our experimental evaluation, the top four maps are 8×8 cells in size, while the bottom four are 12×12. For each size, we employed maps with a varying number of starting bases and workers, resulting in games with different average branching factors, and average lengths. *max* always starts at the top, and *min* at the bottom; each player start with 5 resources (the number displayed on each player base); and each resource mine has 20 resources available.

- Four 8x8 maps (**8x8-1base**, **8x8-2base**, **8x8-3base**, and **8x8-4base**), shown in the top half of Figure 5. In the simplest of them (**8x8-1base**), each player starts with one base and one worker, and near a single resource mine. In the most complex of them (**8x8-4base**), each player starts with four bases and four workers, right next to a row of 7 resource mines.

- Four 12x12 maps (**12x12-1base**, **12x12-2base**, **12x12-3base**, and **12x12-4base**), shown in the bottom half of Figure 5, analogous to the 8x8 maps, but where players start further apart (given the larger dimensions), thus increasing the average length of a game and allowing a larger maximum number of units in the map.

We performed two experiments to assess the performance of each of the CMAB sampling strategies described above when used in MC and MCTS in the context of RTS games:

- **Branching Factor Analysis**: to measure the complexity of the games in each of the 8 maps (and thus compare the results obtained here to those reported in *Experiment 1*, see Section 6.1), average game length and branching factor in each map were analyzed.

- **Round-Robin Analysis**: we selected some of the top performing configurations from the previous experiment, and we ran a round-robin tournament where each configuration played against all others in all the different maps.

Games were limited to 3000 cycles, after which the game was considered a draw. Moreover, in the MC and MCTS implementations in μRTS, when the computation budget is set

Table 3: Median, Average and Maximum branching factor encountered in each of the eight maps during Experiment 2. * some branching factor calculations timed out, median/average/max taken of the ones that did not time out.

Map	Branching			Game Length	
	Median	Average	Max	Average Cycles	Average Decision Cycles
8x8-1base	14.50	1466.35	2.65×10^5	428.33	48.93
8x8-2base	84.00	1.87×10^6	2.49×10^9	474.33	58.93
8x8-3base	106.00	4.53×10^5	1.75×10^8	380.33	52.10
8x8-4base	342.50	9.52×10^5	3.74×10^8	267.00	38.60
12x12-1base	30.50	1.23×10^6	6.14×10^8	972.67	158.17
12x12-2base	112.00	1.56×10^{12}	2.90×10^{15}	893.27	151.00
12x12-3base*	546.75	1.60×10^{15}	4.98×10^{18}	804.00	135.3
12x12-4base*	15330.00	1.08×10^{17}	8.07×10^{19}	777.80	126.13

to T playouts, this does not mean that each decision is made by running MC or MCTS for T playouts. Instead, what this means is that each bot has a computation budget of T playouts per game cycle. Thus, in situations where a bot does not need to issue an action during a few cycles in a row (e.g., because all of its units are already busy), a bot can launch an execution of MC or MCTS that spreads over several game cycles (i.e., the bot starts a search process in the first game cycle, and continues the search during the subsequent game cycles until it needs to produce an action). The following subsections describe the results of each of the experiments.

8.2 Experiment 3: Branching Factor Analysis

We recorded the branching factor from the point of view of both players by making an AI that uses MC search (with a computation budget of 1000 playouts per frame and **NS**(0.4,0.33,0) as the CMAB strategy) play five games against each of three of the scripted AIs that come with μRTS (**RandomBiased**, **LightRush** and **WorkerRush**) in each one of the 8 maps (a total of $3 \times 8 \times 5 = 120$ games). The scripted AI always played as player *max*, and the MC AI always played as player *min*. We selected these different scripted AIs, just to have a variety of games, in order to gather a variety of game states to estimate the branching factor from.

Table 3 shows the median, average, and maximum branching factors encountered during this experiment for each of the eight maps. The right-hand side of Table 3 shows the average game length in terms of both game cycles, and "Decision Cycles" (where a "Decision Cycle" is a game cycle where there was at least one idle unit for which a player had to produce an action). We can see that in the simplest map (**8x8-1base**, used in the past for evaluation of different CMAB strategies, see Ontañón, 2013 and Shleyfman et al., 2014), branching factors do not grow very large (the average is 1466.35, with a median of 14.50). However, as we increase the number of bases, and especially if we use the larger 12x12 maps, the branching factor grows very rapidly. In the extreme cases of the **12x12-3base** and **12x12-4base** maps, some of the branching factors were too large to be computed in a reasonable amount of time. For example, in the **12x12-4base** map, the average branching factor for
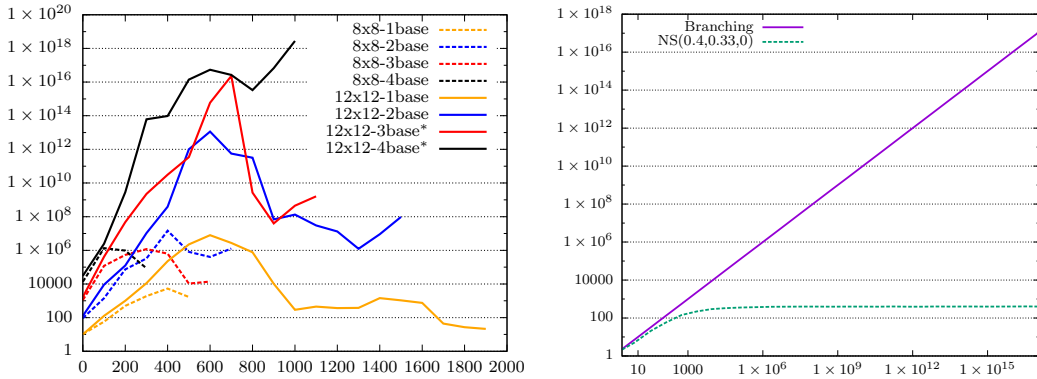
Figure 6: **Left**: average branching factor over the course of games in each of the 8 maps; The horizontal axis represents time (in game cycles), and the vertical axis is the average branching factor. **Right**: comparison of the number of macro-arms explored by **NS**(0.4,0.33,0) with a budget of 1000 playouts as the branching factor grows. * some branching factor calculations timed out, average taken of the ones that did not time out.

the states where we could actually compute it (we set a timeout of 2 hours of CPU time to calculate the branching factor of a game state) was 1.08×10^{17}, and median 15330.00 (which means that half of the times the branching factor was larger than 15330.00). In 12x12 maps, the branching factor is smaller than 1000 66.55% of the times, it is between 1000 and one million 19.06% of the times, and it is larger than one million 14.39% of the times. The left-hand side of Figure 6 shows the average branching factor over time for each of the 8 maps. We can see that branching factor starts small at the beginning, when there are few units in the map, and then grows very rapidly. Branching factor tends to decrease toward the end game, since players destroy each other's units during the game. Moreover, the right-hand side of Figure 6 shows the number of macro-arms explored by **NS**(0.4,0.33,0) with a budget of 1000 playouts as the branching factor grows. As can be seen, **NS**(0.4,0.33,0) flattens out at exploring about 400 macro-arms (corresponding to $\epsilon_0 = 0.4$). In average, **NS**(0.4,0.33,0) explored 36.43% of all the possible macro-arms (notice that even if the percentage of explored macro-arms is nearly 0 for those game states with large branching factor, 66.55% of those have branching factor smaller than 1000).

Finally, in the Java μRTS implementation of these algorithms, running MC search takes about 271ms to run 1000 playouts in 8x8 maps, and about 615ms in 12x12 maps on an Intel Core i7 3.1GHz (the sampling strategy makes little difference, since the main bottleneck is running the forward model to run the playouts). In an optimized C++ implementation, we thus estimate that between 200 and 1000 playouts would be feasible to be run on real time (10 to 24 frames per second) in a commercial RTS game depending on the complexity of the game state.
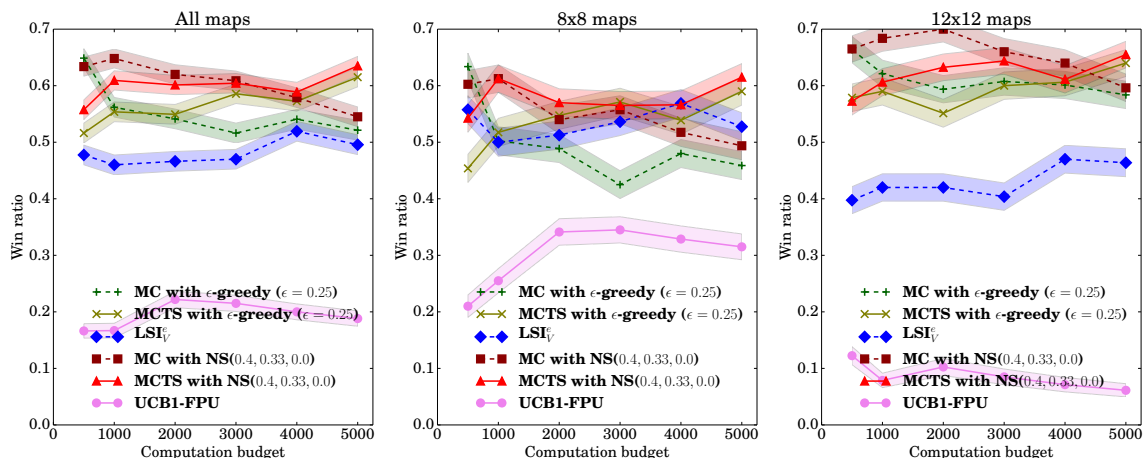
Figure 7: Win ratio of different approaches when doing a round-robin tournament with different computation budgets: averaged over all maps (left), only the 8x8 maps (center) and only the 12x12 maps (right).

8.3 Experiment 4: Round-Robin Analysis

We performed a series of round-robin tournaments involving six different AIs:

- **MC with ϵ-greedy** ($\epsilon = 0.25$): Monte Carlo search using ϵ-greedy as the tree policy.

- **MCTS with ϵ-greedy** ($\epsilon = 0.25$): MCTS using ϵ-greedy as the tree policy.

- **MC with LSI_V^e** (Shleyfman et al., 2014): plain Monte Carlo search using LSI_V^e. Notice that LSI cannot be used with MCTS, since it needs to know in advance the computation budget to be used in a given node of the tree.

- **MC with NS**$(0.4, 0.33, 0.0)$: Monte Carlo search using naïve sampling, NS$(0.4, 0.33, 0.0)$, as the sampling policy.

- **MCTS with NS**$(0.4, 0.33, 0.0)$: Monte Carlo Tree Search using NS$(0.4, 0.33, 0.0)$ as the tree policy.

- **UCB1-FPU**: Monte Carlo Tress Search using a UCB1 sampling strategy with an FPU constant set to: $FPU = f \times \rho(s) + (1 - f)$, where $\rho(s)$ is the value of the evaluation function applied to the game state in the current game state, and $f = 0.95$, set empirically (intuitively, this basically sets the FPU constant to slightly higher than the value of the current game state, so that if actions are found early that improve the evaluation function, those are explored right away).

Notice that we test both ϵ-greedy and naïve sampling using an MC and a MCTS search algorithm, in order to separate the performance that comes from the sampling strategy from the performance that comes from the search algorithm.

In each round-robin tournament, each AI played 20 games against each other AI (10 times as player *max*, and 10 times as player *min*) in each of the 8 maps, resulting in a total of 2400 games per tournament. We played six such round-robin tournaments, using a computation budget of 500, 1000, 2000, 3000, 4000 and 5000 playouts per game cycle respectively.

The left hand side of Figure 7 shows the win ratio (vertical axis) of each of the AIs as a function of the computational budget used in each of the tournaments (horizontal axis). "Win ratio" was calculated as the average score that bots got in each game, scoring 1 for winning, 0.5 for drawing, and 0 for losing the game. The bands around the plots represent the 95% confidence interval in the win ratio. Thus, when the bands of two plots do not overlap, their difference is statistically significant with $p = 0.05$. As can be seen, **MC with ϵ-greedy** won the 500 playouts tournament (but with a difference that is not statistically significant), **MC with NS**$(0.4, 0.33, 0.0)$ won the 1000, 2000 and 3000 tournaments (the 1000 and 2000 playout tournaments with a statistically significant difference with respect to the ϵ-greedy strategies), and **MCTS with NS**$(0.4, 0.33, 0.0)$ won the 4000 and 5000 (but with a difference that is not statistically significant).

Looking at the left-hand side of Figure 7 closely, we can see that except for the extreme case of budget 500, naïve sampling dominates the tournaments for low computation budgets (1000, 2000), and as the computation budget increases, the other strategies catch up. The exception is the case of budget 500, where ϵ-greedy seems to work very well. After close inspection of the results, we noticed that this could be caused by two separate facts. First, the exploration constant used by ϵ-greedy ($\epsilon = 0.25$) was better suited for this low-computation budget setting than the higher exploration constants used by naïve sampling in our experiments. We set all of these values experimentally based on overall performance across all tournaments. Second, we noticed that the performance gain of naïve sampling with respect to ϵ-greedy was smaller when using MCTS than when using MC, and in the special case of budget 500, MCTS with naïve sampling seems to work very poorly. This makes us formulate the hypothesis that for very low computation budgets, the estimation of the rewards of the individual unit-actions made by the local MABs is not reliable, and thus, it does not help the search process. We will verify this hypothesis in the next section.

Another interesting result we observe from Figure 7 is that MC dominates MCTS for low computation budgets (all the MC bots are displayed with dashed lines, and the MCTS bots with solid lines), but as the computation budget increases, MCTS outperforms MC. This is observed both for ϵ-greedy and for naïve sampling. We would like to point out, however, that we set the exploration constants of the sampling strategies based on the performance of the MC AIs. Thus, it is possible that different exploration constants could make the MCTS AIs perform better.

Finally, UCB1-FPU performed very poorly in this experiment, as in our previous experiments. Specially in the 12x12 maps. In the 8x8 maps, the performance was closer to the other policies, since the branching factors were smaller. Although FPU helped UCB1 significantly in our preliminary experiments (without adding an FPU constant, UCB1's performance is even lower), we found that finding the correct value for the FPU constant was not trivial. In the results presented in Figure 3, we fine-tuned this constant for each individual CMAB. But in real play, the FPU constant needs to be set automatically for each game situation. A fixed FPU constant seemed not to work, and we employed the
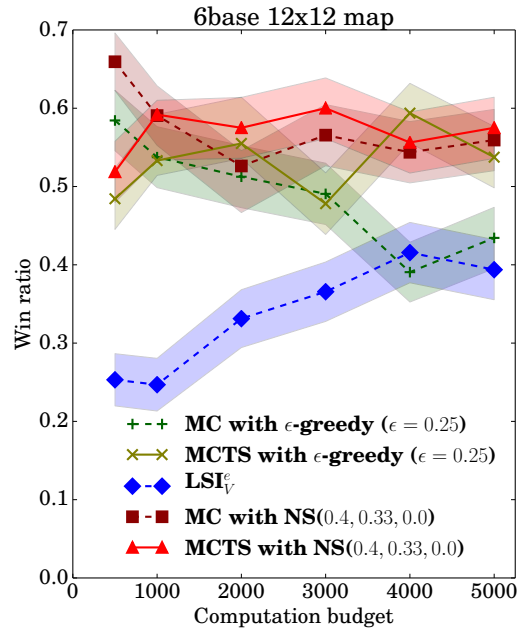
Figure 8: Win ratio of different approaches when doing a round-robin tournament with different computation budgets on a complex 12x12 map, where each player starts with six bases and six workers.

dynamic scheme $FPU = f \times \rho(s) + (1 - f)$ described above, which worked the best in our experiments, but still underperforms compared to the other strategies. However, comparing the results from Figures 3 and 7, we believe there is still room for improvement with better FPU constant setting procedures.

In order to get better insight into these results, the center and right plots of Figure 7 show these results considering only the 8x8 maps (center) and only the 12x12 maps (right). Consistent with the results reported earlier in this paper, for the maps with larger branching factors, naïve sampling performs better. Moreover, we can see that the crossover point between MC and MCTS occurs later for maps with larger branching factors.

Finally, in order to confirm the advantage of naïve sampling over the other sampling strategies on larger maps, we performed an additional set of experiments on a more complex map (12x12 map, where each player started with 6 bases and 6 workers). Results are shown in Figure 8. Since this figure shows results for only one map, in order to reduce uncertainty, we made each AI play 40 times against each other, instead of 20 as in the previous plots. The results show that both naïve sampling AIs consistently achieve higher win ratios than the other AIs, with the only other AI with comparable (but lower in most cases) performance being MCTS with ϵ-greedy. In these experiments, the performance of MC with ϵ-greedy degraded quickly compared with the other strategies. Moreover, given that the results shown in Figure 8 correspond to only one map, we see a larger overlap of the 95% confidence intervals in the figure, meaning that not all differences are statistically significant. Moreover UCB1-FPU is not shown since it achieved a win ratio of 0.
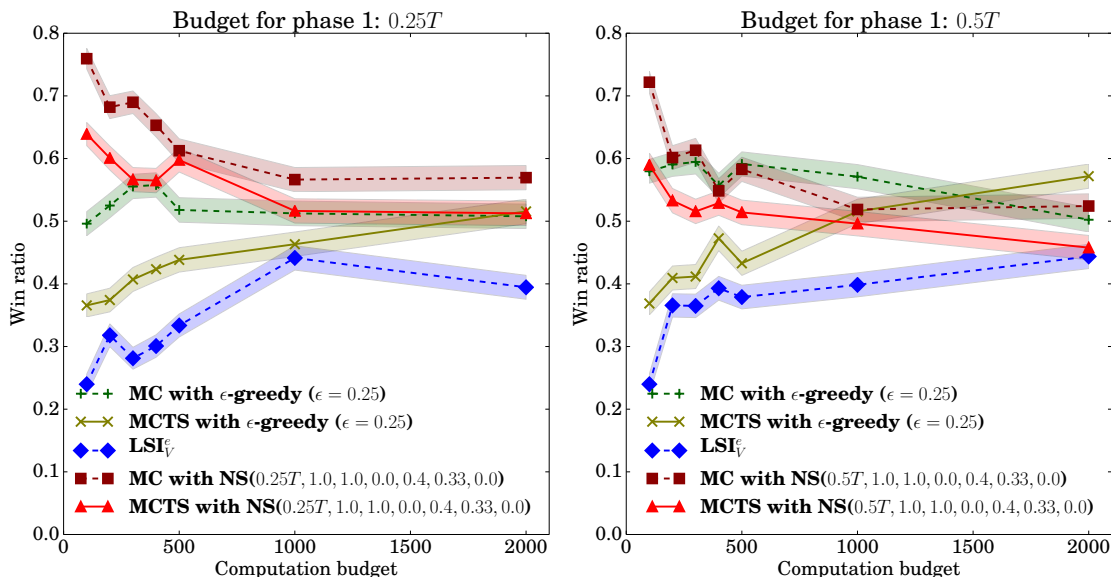
Figure 9: Win ratio of different approaches when doing a round-robin tournament, using two-phase naïve sampling with low computation budgets averaged over all maps.

In conclusion, we can see that different strategies are better suited for different settings: when the computational budget is very low, MC seems to outperform MCTS, while as the computation budget increases this is reversed. Also, for low computational budgets ϵ-greedy seems to achieve very good results (the next experiment will look at this more closely), and as the computation budget increases, naïve sampling dominates (especially for situations with large branching factors). Finally, as the computation budget increases even further, then the result from different sampling strategies seems to converge (ϵ-greedy, LSI and naïve sampling using MC tend to converge to the same value, and ϵ-greedy and naïve sampling using MCTS also tend to converge).

8.4 Experiment 5: Round-Robin with Small Budget

The previous experiment showed that an ϵ-greedy strategy combined with MC seemed to work very well for very low computation budgets (500 playouts per decision cycle). Upon close inspection of the results, we observed that naïve sampling struggled with low computation budgets, since in the first iterations, the local MABs still do not have meaningful estimations, and thus cannot guide exploration. In order to verify this hypothesis, we experimented with two-phase naïve sampling strategies that would spend initially more time exploring (in order to get a good estimation in the local MABs) and only then start exploiting. Specifically, we repeated the round robin experiments, focusing on computation budgets between 100 to 2000, removing the **UCB1-FPU** bot which did not perform well in the previous experiment, and using the following versions of naïve sampling:

- **NS**$(0.25T, 1.0, 1.0, 0.0, 0.4, 0.33, 0.0)$: during the first 25% of the computation budget, this strategy will basically select macro-arms randomly, and then during the remaining

75% of the computation budget, it will run the same parameter configuration of naïve sampling as used in Experiment 4.

- **NS**$(0.5T, 1.0, 1.0, 0.0, 0.4, 0.33, 0.0)$: the same, but exploring randomly during the first 50% of the computation budget.

Figure 9 shows the experimental results. The first thing we see is that using 25% of the computation budget to explore random macro-arms (left-hand side of Figure 9) seems to work much better than using 50% of the computation budget (right-hand side of Figure 9). For example, using MC with a two-phase NS strategy achieves a win ratio of over 60% with computation budgets lower than 500 in the 25% exploration setting. This shows that when using very low computation budgets with naïve sampling, it is important to dedicate some amount of the computation budget initially for exploration.

We performed preliminary experiments with larger computation budgets (not reported) and observed that for higher computation budgets, two-phase strategies either did not seem to make a difference or performed worse than standard naïve sampling. So, two-phase sampling seems to only help in cases of low computation budgets.

9. Related Work

Several areas are related to the work presented in this paper: combinatorial multi-armed bandits (CMABs), AI techniques for RTS games, as well as more general work on multiagent planning or decentralized sequential decision making. While existing work on CMABs is covered in Section 5, this section briefly discusses work on the other related areas, as well as their connection with the work presented in this paper.

Since the first call for research on RTS game AI by Buro (2003), a wide range of AI techniques have been explored to play RTS games. For example, reinforcement learning (RL) has been used for controlling individual units (Marthi, Russell, & Latham, 2005; Jaidee & Muñoz-Avila, 2012), groups of units (Wender & Watson, 2012; Usunier, Synnaeve, Lin, & Chintala, 2016), and even to make high-level decisions in RTS games (Sharma, Holmes, Santamaría, Irani, Isbell Jr, & Ram, 2007). The main issue when deploying RL in RTS games is computational complexity, as the state and action space are very large. The aforementioned techniques address these problems by either focusing on individual units, small-scale combat, or by using domain knowledge to abstract the game state in order to simplify the problem. Although recent approaches are starting to scale up to larger and larger combat situations (Usunier et al., 2016) by using techniques such as deep reinforcement learning, they are still far from scaling all the way up to full-game play.

Other machine learning techniques, such as case-based approaches for learning to select among high-level strategies (Aha, Molineaux, & Ponsen, 2005), or for learning to choose the right "build-order" (Weber & Mateas, 2009), have also been proposed. In our previous work, we showed that *learning from demonstration* (Ontañón, Mishra, Sugandh, & Ram, 2007, 2010), although promising, also struggles to generalize due to the large variety of situations that can arise in RTS games.

More related to the work presented in this paper, there has also been a significant amount of work to design game-tree search approaches that can handle RTS games. Early work used plain Monte Carlo search (Chung et al., 2005), but very soon, work shifted to Monte

Carlo Tree Search (Balla & Fern, 2009). And, as mentioned in Section 2.2, techniques now exist that can perform game tree search in domains with durative actions (Churchill et al., 2012), or simultaneous moves (Kovarsky & Buro, 2005; Saffidine et al., 2012), both features of RTS games. Moreover, most recent work has focused on addressing the problem of the large branching factors present in RTS games. While in this paper we focused on a bandit strategy that can handle the combinatorial branching factor on RTS games, other work to address this problem can be categorized around four main lines:

- *Game state abstractions*: the idea is to re-represent the game state, removing some of the low-level details, in order to make the search space smaller. For example, Balla and Fern (2009) represented the game state by clustering units into groups, idea which has been expanded upon in later work (Justesen, Tillman, Togelius, & Risi, 2014; Uriarte & Ontañón, 2014).

- *Portfolio approaches*: rather than re-representing the game state, portfolio approaches reduce the combinatorial branching factor of RTS games by only letting the AI pick amongst a predefined fixed set of scripts, rather than having to select among all the possible low-level actions. The first such work was proposed by Chung et al. (2005), but modern versions of the approach perform either greedy search (Churchill & Buro, 2013) or MCTS (Justesen et al., 2014).

- *Hierarchical search*: another idea that has been explored recently is that of performing search at several levels of abstraction, considering high-level decisions first, which condition the potential number of low-level decisions that can be taken. An example of this approach is the idea of *Adversarial HTN planning* (Ontañón & Buro, 2015), which combines minimax search with HTN planning. Another example is the work of Stanescu et al. (2014), who perform game tree search at two separate levels of abstraction, one informing the other.

- Finally, a recent line of work, inspired by the success of *AlphaGO* (Silver, Huang, Maddison, Guez, Sifre, van den Driessche, Schrittwieser, Antonoglou, Panneershelvam, & Lanctot, 2016) has started to explore the idea of integrating machine learning into game tree search. For example, Stanescu, Barriga, Hess, and Buro (2016) trained deep neural networks to automatically learn evaluation functions in μRTS, showing they could significantly outperform the base evaluation function. Another example is our previous work (Ontañón, 2016), where we employed a very similar approach to *AlphaGO*, but using Bayesian models rather than neural networks, in order to inform the search in MCTS.

Notice, however, that all four of these lines of work are orthogonal to the work presented in this paper, and, as a matter of fact, some of these strategies have been explored in conjunction with naïve sampling.

Another related area is that of *multiagent planning* (Durfee, 2001; De Weerdt, Ter Mors, & Witteveen, 2005; Brafman & Domshlak, 2008), which focuses on the problem of automated planning in domains where there are more than one agent performing actions. Of particular relevance to the work presented in this paper is the setting where planning is

centralized and only execution is decentralized. A common framework to model this setting is that of *decentralized Markov Decision Processes* (DEC-MDPs and DEC-POMDPs) (Oliehoek & Amato, 2016). A DEC-POMDP is a Partially Observable Markov Decision Process (POMDP), where there is more than one agent that can act simultaneously, each of them with their own partial perception of the world. All the agents in a DEC-POMDP attempt to maximize the same reward function. The particular case where the joint perception of all the agents corresponds to the complete state (i.e., when considering the joint perception makes the problem fully-observable) is called a DEC-MDP. Algorithms to address DEC-MDPs and DEC-POMDPs have been proposed in the literature, from systematic search, to dynamic programming (Hansen, Bernstein, & Zilberstein, 2004), or approximate algorithms such as best-response approaches (Nair, Tambe, Yokoo, Pynadath, & Marsella, 2003) (which fix the policy of all agents but one, and iteratively each agent optimizes her policy with respect of the fixed policy of the others). The specific RTS-games setting we consider in this paper corresponds to a generalization of DEC-POMDPs, where different agents have different reward functions called *partially observable stochastic games* (POSG) (Hansen et al., 2004) (specifically, in our setting, all the units in an RTS game are divided among the two players, each player having her own reward function). The naïve assumption made by naïve sampling is also connected to the related idea of factored MDPs, where agents approximate the global reward function as a linear combination of local value functions (Guestrin, Koller, & Parr, 2001).

10. Conclusions

Real-time strategy (RTS) games pose a significant challenge to game tree search approaches due to the very large branching factors they involve. In this paper, we explored the possibility of modeling RTS game situations as *combinatorial multi-armed bandit* (CMAB) problems, and study the theoretical and practical behavior of a new family of sampling strategies called naïve sampling. We compare these sampling strategies against other sampling strategies in the literature for CMABs in the context of μRTS.

As our results indicate, for situations with small branching factors, naïve sampling performs similar to other sampling strategies such as LSI or ϵ-greedy. However, as the branching factor grows, the performance of other strategies degrades compared to naïve sampling, especially under tight computation budgets, which is especially relevant for real-time games (as showed in Section 8.2, computation budgets in the order 1000 playouts per game cycle are to be expected).

As part of our future work we would like to better understand the behavior of naïve sampling in the context of MCTS, where the computation budget that can be used in the inner nodes of the tree is smaller the deeper the node is in the tree. As the results in Experiments 4 and 5 showed, for very low computation budgets, the estimations performed by the local MABs might not be accurate enough to be reliable, and thus, two-phase strategies are more appropriate. However, how to decide when to use a two-phase versus a single-phase strategy is still unclear. Additionally, we are currently looking at sampling strategies that incorporate prior knowledge of the domain (such as the neural network models used by AlphaGO, see Silver et al., 2016) into naïve sampling strategies (initial results on this direction indicate that significant gains can be achieved, see Ontañón, 2016). We would

also like to investigate better sampling strategies for CMABs by studying the relation of CMABs with other combinatorial optimization problems, and by studying the relation of naïve sampling to sampling policies for continuous-valued bandits. Finally, we would like to apply naïve sampling-based MCTS approaches to address large-scale RTS games such as *StarCraft*.

Appendix A.

This section contains proofs to the propositions presented earlier, preceded by the simple regret analysis of ϵ-greedy, which will set the basis for some of the proofs.

A.1 Simple Regret Analysis of ϵ-greedy

Assume a MAB with K arms, i.e., a single variable X that can take values $\{v_1, ..., v_K\}$, and where $\overline{\mu}_t(v_i)$ is the average reward estimated for arm i at iteration t.

Assume also that the expected difference in expected reward between the optimal arm and a non-optimal arm is $D = E_{v_i \in \mathcal{X}^-}(\mu^* - \mu(v_i))$, where μ^* is the expected reward of an optimal arm, and $\mathcal{X}^- = \{v_i \in \mathcal{X} | \mu(v_i) < \mu^*\}$ is the set of non-optimal arms; and that the minimum difference between the optimal arm and a non-optimal arm is at least $d = min_{v_i \in \mathcal{X}^-}(\mu^* - \mu(v_i))$.

Given a suboptimal arm v_i and an optimal arm v^* that have been sampled at least m times, the probability that $\overline{\mu}_t(v_i) \geq \overline{\mu}_t(v^*)$ can be bounded in the following way. By Hoeffding's inequality (Hoeffding, 1963), we know that the probability that the empirical estimation $\overline{S}$ of the mean of a variable S differs in more than d from the actual mean $E[\overline{S}]$ after having sampled it m times, is bounded by $P(E[\overline{X}] - \overline{S} \geq d) \leq e^{-2d^2 m}$. Now let S be the difference in observed reward between v^* and v_i. Then the empirical estimate $\overline{S}$ is $(\overline{\mu}_t(v^*) - \overline{\mu}_t(v_i))$, and by assumption we know that $E[\overline{S}] = \Delta \geq d$. Thus, by Hoeffding's inequality, we know that:

$$P(\Delta - (\overline{\mu}_t(v^*) - \overline{\mu}_t(v_i)) \geq \Delta) \ \leq \ e^{-2\Delta^2 m} \ \leq \ e^{-2d^2 m}$$

And thus:

$$
\begin{aligned}
P(\overline{\mu}_t(v_i) \geq \overline{\mu}_t(v^*)) \ &= \ P((\overline{\mu}_t(v^*) - \overline{\mu}_t(v_i) \leq 0) \\
&= \ P(d - (\overline{\mu}_t(v^*) - \overline{\mu}_t(v_i)) \geq d) \\
&\leq \ e^{-2d^2 m}
\end{aligned}
$$

Let v_t^{best} be the arm believed to be the best at iteration t (the one with the highest estimated reward so far), $v_t^{best-no}$ be the arm from the set of non-optimal arms $\mathcal{X}^-$ that at iteration t has the highest estimated reward so far, and n_t^{best} and $n_t^{best-no}$ the number of times these arms have been sampled respectively. Now, let us assume that there is a single optimal arm v^* (which has been sampled n_t^* times). Now, notice that if $\mu_t(v_t^{best-no}) > \overline{\mu}_t(v^*)$, then v_t^{best} will be $v_t^{best-no}$ instead of v^*, and thus the arm believed to be the best at iteration t is not the optimal one. Therefore, we can estimate the probability of v_t^{best} not to be the optimal arm as:

$$P(\overline{\mu}_t(v_t^{best-no}) \geq \overline{\mu}_t(v^*)) = P((\overline{\mu}_t(v^*) - \overline{\mu}_t(v_t^{best-no})) \leq 0)$$
$$\leq e^{-2d^2 min(n_t^{best-no}, n_t^*)}$$
$$\leq e^{-2d^2 \frac{\epsilon}{K} t}$$

If we lift the assumption that there is a single optimal arm, then the probability of v_t^{best} not to be optimal is even lower, and thus this bound still applies. Thus, the probability of choosing the wrong arm after t iterations decreases exponentially. The regret for ϵ-greedy is:

- *Instantaneous regret*:
$$r'_t \approx \left(\epsilon \frac{K^{no}}{K} + (1 - \epsilon)e^{-2d^2 \frac{\epsilon}{K} t} \right) D$$

  where $K^{no} < K$ is the number of non-optimal arms. Thus, when $t \to \infty$, this tends to $\epsilon \frac{K^{no}}{K} D$, which is a constant; and when K is very large (and $K^{no}/K \approx 1$), can be simplified as ϵD.

- *Cumulative regret*: since the instantaneous regret tends to a constant, cumulative regret is linear $R_t = O(t\epsilon D)$

- *Simple regret*: the simple regret decreases exponentially:
$$r_t \approx De^{-2d^2 \frac{\epsilon}{K} t}$$

Moreover, notice that decreasing ϵ reduces the instantaneous and cumulative regret but increases the simple regret. $\epsilon = 1$, which achieves the highest expected instantaneous and cumulative regret, achieves the lowest expected simple regret.

A.2 Propositions and Proofs

Proposition 1: The *cumulative regret* of NS($\epsilon_0, \epsilon_l, \epsilon_g$) grows linearly as $R_T = O((1 - p^*)DT)$, where T is the number of iterations, D is the expected difference in expected reward between an optimal macro-arm and a non-optimal macro-arm, and $p^* \geq (1 - \epsilon_0)(1 - \epsilon_g)$ is the probability of selecting an optimal macro-arm when $T \to \infty$.

Proof: Formally, $D = E_{V_i \in \mathcal{X}^-}(\mu^* - \mu(V_i))$, where $\mathcal{X}^- = \{V_i \in \mathcal{X}|L(V_i) \wedge \mu(V_i) < \mu^*\}$ is the set of legal non-optimal macro-arms.

The probability of selecting an optimal macro-arm V^* when $T \to \infty$ can be calculated as follows. When $T \to \infty$, we can assume that each macro-arm has been sampled enough times as for V^* having higher reward than all the non-optimal macro-arms. Thus, NS($\epsilon_0, \epsilon_l, \epsilon_g$) would select it in the following circumstances:

- When exploiting, V^* will be selected with probability $(1-\epsilon_g)+\epsilon_g/N$ if there is a single optimal macro-arm, and higher if there is more than one optimal macro-arm (where N is the number of legal macro-arms). Assuming that N is large, this is approximately $(1 - \epsilon_g)$.

- When exploring, V^* will be selected with probability at least $\prod_{i=1...n}((1-\epsilon_l)+\epsilon_l/K_i)$ (the probability will be higher if there is more than one optimal macro arm).

Thus, for large N the probability of selecting an optimal macro-arm V^* is at least: $p^* = \epsilon_0 \left(\prod_{i=1\ldots n} ((1 - \epsilon_l) + \epsilon_l/K_i) \right) + (1 - \epsilon_0)(1 - \epsilon_g)$, from which we know that $p^* \geq (1 - \epsilon_0)(1 - \epsilon_g)$ (and if the number of variables n is large, $\epsilon_l > 0$, and there are few optimal macro-arms, the inequality will be tight).

Thus, the instantaneous regret of NS($\epsilon_0, \epsilon_l, \epsilon_g$) will be $r'_T \approx (1 - p^*)D$, which implies that the cumulative regret will be $r_T = O((1 - p^*)DT)$. □

Proposition 2: The *simple regret* of NS($\epsilon_0, \epsilon_l, \epsilon_g$) decreases at an exponential rate as $r_T = O(De^{-2d^2 T p_i})$, where $p_i \geq \epsilon_0 \Pi_{j=1\ldots n} \frac{\epsilon_l}{K_j} + (1 - \epsilon_0)\frac{\epsilon_g}{N}$, D is as in Proposition 1, and d is the minimum difference between an optimal macro-arm and a non-optimal macro-arm.

Proof. Formally, $d = min_{V_i \in \mathcal{X}^-}(\mu^* - \mu(V_i))$, where $\mathcal{X}^-$ is as in Proposition 1. Given a suboptimal macro-arm V_i and an optimal macro-arm V^*, after a sufficiently large number of iterations T:

- The probability of selecting V_i is $p_i \geq \epsilon_0 \Pi_{j=1\ldots n} \frac{\epsilon_l}{K_j} + (1 - \epsilon_0)\frac{\epsilon_g}{N}$, and thus, we can expect the arm to have been selected at least $n_T = Tp_i$ times.

- The probability of selecting an optimal macro-arm V^* is $p^* \geq (1 - \epsilon_0)(1 - \epsilon_g)$, and thus, we can expect the arm to have been selected at least $n_T^* = Tp^*$ times.

Now, given V_t^{best} to be the best macro-arm at iteration t, the probability that V_t^{best} is not an optimal arm is (using Hoeffding's inequality, see Hoeffding, 1963):

$$P(\overline{\mu}_t(V_t^{best}) \geq \overline{\mu}_t(V^*)) \;\; = \;\; P((\overline{\mu}_t(V^*) - \overline{\mu}_t(V_t^{best})) \leq 0)$$
$$\leq \;\; e^{-2d^2 min(n_t, n_t^*)}$$

Assuming $\epsilon_0 \leq 0.5$, $\epsilon_g \leq 0.5$ and $\epsilon_l \leq 0.5$, and $N \gg 2$, p_i is expected to be lower than p^*, and thus:

$$P((\overline{\mu}_t(V^*) - \overline{\mu}_t(V_t^{best})) \leq 0) \leq e^{-2d^2 n_t}$$

Thus, if the expected difference in expected reward between the optimal arm and a non-optimal arm is D, after a sufficiently large number of iterations T, the expected simple regret $r_T \approx De^{-2d^2 n_T}$. □

Proposition 3: In a CMAB with n variables, the probability that after t iterations using a NS($\epsilon_0, \epsilon_l, \epsilon_g$) sampling strategy an optimal macro-arm V^* has not been explored at least once, decreases exponentially as a function of t, and is at most $(1 - p)^{t\epsilon_0}$, where $p = \prod_{i=1\ldots n}(\epsilon_l/K_i)$.

Proof. The expected number of exploration iterations done after t iterations is $m = t\epsilon_0$. When exploring, the probability that a given value v_i^j is selected for variable X_i at a given exploration iteration, is at least ϵ_l/K_i (it could be higher, if v_i^j happens to be the value with the highest expected reward so far). Thus, the probability of selecting $V^* = (v_1^*, \ldots, v_n^*)$ during exploration is at least $p = \prod_{i=1\ldots n}(\epsilon_l/K_i)$. Therefore, the probability of not selecting V^* after m exploration iterations is at most $(1 - p)^m$, i.e., it decreases exponentially as a function of t. □

Proposition 4: The *cumulative regret* of $\text{NS}(k, \epsilon_0^1, \epsilon_l^1, \epsilon_g^1, \epsilon_0^2, 0, \epsilon_g^2)$ when k is a constant grows linearly when $T \gg k$:

$$R_T = O\left(t\left[(1 - \epsilon_g - q_k + \epsilon_g q_k)d + \epsilon_g D\right]\right)$$

where $q_k = 1 - (1-p)^{k\epsilon_0^1}$, and $p = \prod_{i=1...n}\left(\epsilon_l^1 / K_i\right)$.

Proof. Let us use D and d as before (difference between an optimal macro-arm and a random macro-arm, and difference between the best non-optimal macro-arm and an optimal macro-arm respectively). Notice that if $\epsilon_0^2 = 0$, it means that the second stage is just an ϵ-greedy strategy with the set of explored macro-arms with parameter ϵ_g. Additionally, according to Proposition 3, with probability at least $q_k = 1 - (1-p)^{k\epsilon_0^1}$, an optimal macro-arm will be explored during the first phase.

Therefore, with probability q_k, for $T \gg k$, an optimal macro-arm will be in the global MAB. Also, if $T \gg k$, the number of times the macro-arms in the global MAB have been sampled during the first k iterations will be negligible. Thus we can just focus on the second stage. Therefore, with probability q_k we will have a cumulative regret $O(t\epsilon_g D)$.

On the other hand, with probability $(1 - q_k)$, the instantaneous regret will be at least d (since no optimal macro-arm is in the global MAB, and thus, the difference between the best macro-arm in the global MAB and an optimal macro-arm is at least d). In this case, instantaneous regret will converge to a higher constant: $r_t' \approx aD + (1-a)d$, where $a = \left(\epsilon_g \frac{K^{no}}{K} + (1 - \epsilon_g)e^{-2d^2 \frac{\epsilon_g}{K}t}\right)$. When $t \to \infty$, and K is large, this converges to: $r_t' \approx d + \epsilon_g(D - d)$, which gives a cumulative regret $O\left(t(d + \epsilon_g(D - d))\right)$

Thus, the expected cumulative regret of two-phase naïve when k is a constant and $\epsilon_l^0 = 0$ is (after rearranging):

$$R_t = O\left(t\left[(1 - \epsilon_g - q_k + \epsilon_g q_k)d + \epsilon_g D\right]\right)$$

$\square$

Proposition 5: The *simple regret* of $\text{NS}(k, \epsilon_0^1, \epsilon_l^1, \epsilon_g^1, \epsilon_0^2, 0, \epsilon_g^2)$ when k is a constant and $T \gg k$ is lower bounded by $(1 - q_k)d$, where $q_k = 1 - (1-p)^{k\epsilon_0^1}$, and d is difference between the best non-optimal macro-arm and an optimal macro-arm.

Proof. Following a similar line of reasoning as for Proposition 4, with probability q_k at least one optimal macro-arm will be in the global MAB and for $T \gg k$ we can just focus on the second stage. Thus, with probability q_k, following the ϵ-greedy analysis presented above, the simple regret will be approximately $De^{-2d^2 \frac{\epsilon_g}{K}T}$.

With probability $(1 - q_k)$, however, there will be no optimal macro-arm in the global MAB. Let d' be the difference in reward between the best macro-arm in the global MAB (V^+) and the second best, and D' the average difference in reward between the best macro-arm in the global MAB and the rest. In this case, the simple regret with respect of not choosing V^+ will be approximately $D'e^{-2d'^2 \frac{\epsilon_g}{K}T}$. However, since the difference in reward between V^+ and an optimal macro arm is at least d, then we need to add d to the actual simple regret.

Thus, we have that the simple regret of two-phase naïve when k is a constant and $\epsilon_l^0 = 0$ is:

$$r_t \approx q_k \left(D e^{-2d^2 \frac{\epsilon_g}{K} T} \right) + (1 - q_k) \left(d + D' e^{-2d'^2 \frac{\epsilon_g}{K} T} \right)$$

Which, when $T \to \infty$, converges to: $(1 - q_k)d$. $\qquad\square$

References

Aha, D. W., Molineaux, M., & Ponsen, M. (2005). Learning to win: Case-based plan selection in a real-time strategy game. In *Case-based reasoning research and development*, pp. 5–20. Springer.

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

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

Brafman, R. I., & Domshlak, C. (2008). From one to many: Planning for loosely coupled multi-agent systems. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS 2008)*, pp. 28–35.

Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., & Colton, S. (2012). A survey of Monte Carlo Tree Search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, *4*(1), 1–43.

Bubeck, S., & Cesa-Bianchi, N. (2012). Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *ArXiv e-prints, arXiv:1204.5721.*

Bubeck, S., Munos, R., & Stoltz, G. (2011). Pure exploration in finitely-armed and continuous-armed bandits. *Theoretical Computer Science*, *412*(19), 1832–1852.

Bubeck, S., Munos, R., Stoltz, G., & Szepesvari, C. (2008). Online optimization in $\mathcal{X}$-armed bandits. In *Twenty-Second Annual Conference on Neural Information Processing Systems*.

Buro, M. (2003). Real-time strategy games: a new AI research challenge. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 2003)*, pp. 1534–1535, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Cazenave, T. (2015). Sequential halving applied to trees. *IEEE Transactions on Computational Intelligence and AI in Games*, *7*(1), 102–105.

Chen, W., Wang, Y., & Yuan, Y. (2013). Combinatorial multi-armed bandit: General framework and applications. In *Proceedings of the 30th International Conference on Machine Learning (ICML 2013)*, pp. 151–159.

Chinchalkar, S. (1996). An upper bound for the number of reachable positions. *ICCA Journal*, *19*(3).

Chung, M., Buro, M., & Schaeffer, J. (2005). Monte carlo planning in RTS games. In *Proceedings of the IEEE Conference on Computational Intelligence in Games (CIG 2005)*.

Churchill, D., Saffidine, A., & Buro, M. (2012). Fast heuristic search for RTS game combat scenarios. In *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment conference (AIIDE 2012)*. The AAAI Press.

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

De Weerdt, M., Ter Mors, A., & Witteveen, C. (2005). Multi-agent planning: An introduction to planning and coordination. In *Handouts of the European Agent Summer*.

Durfee, E. H. (2001). Distributed problem solving and planning. In *Multi-agent systems and applications*, pp. 118–149. Springer.

Feldman, Z., & Domshlak, C. (2014). Simple regret optimization in online planning for Markov decision processes. *Journal of Artificial Intelligence Research*, *51*, 165–205.

Gai, Y., Krishnamachari, B., & Jain, R. (2010). Learning multiuser channel allocations in cognitive radio networks: A combinatorial multi-armed bandit formulation. In *2010 IEEE Symposium on New Frontiers in Dynamic Spectrum*, pp. 1–9.

Gelly, S., & Silver, D. (2007). Combining online and offline knowledge in UCT. In *Proceedings of the 24th International Conference on Machine learning (ICML 2007)*, pp. 273–280. ACM.

Gelly, S., & Wang, Y. (2006). Exploration exploitation in Go: UCT for monte-carlo Go. In *NIPS: Neural Information Processing Systems Conference On-line trading of Exploration and Exploitation Workshop*.

Guestrin, C., Koller, D., & Parr, R. (2001). Multiagent planning with factored MDPs. In *Proceedings of the Conference on Neural Information Processing Systems (NIPS 2001)*, Vol. 1, pp. 1523–1530.

Hansen, E. A., Bernstein, D. S., & Zilberstein, S. (2004). Dynamic programming for partially observable stochastic games. In *Proceedings of the Association for the Advancement of Artificial Intelligence Conference (AAAI 204)*, Vol. 4, pp. 709–715.

Hoeffding, W. (1963). Probability inequalities for sums of bounded random variables. *Journal of the American statistical association*, *58*(301), 13–30.

Jaidee, U., & Muñoz-Avila, H. (2012). Classq-l: A q-learning algorithm for adversarial real-time strategy games. In *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*.

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

Karnin, Z., Koren, T., & Somekh, O. (2013). Almost optimal exploration in multi-armed bandits. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pp. 1238–1246.

Kocsis, L., & Szepesvri, C. (2006). Bandit based monte-carlo planning. In *Proceedings of the European Conference on Machine Learning (ECML 2006)*, pp. 282–293. Springer.

Kovarsky, A., & Buro, M. (2005). Heuristic search applied to abstract combat games. In *Canadian Conference on AI*, pp. 66–78.

Kuhn, H. W. (1955). The hungarian method for the assignment problem. *Naval research logistics quarterly*, *2*(1-2), 83–97.

Marthi, B., Russell, S., & Latham, D. (2005). Writing stratagus-playing agents in concurrent ALisp. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 2005)*.

Nair, R., Tambe, M., Yokoo, M., Pynadath, D., & Marsella, S. (2003). Taming decentralized POMDPs: Towards efficient policy computation for multiagent settings. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 2003)*, pp. 705–711.

Oliehoek, F. A., & Amato, C. (2016). *Concise Introduction to Decentralized POMDPs*. Springer.

Ontañón, S., Mishra, K., Sugandh, N., & Ram, A. (2010). On-line case-based planning. *Computational Intelligence*, *26*(1), 84–119.

Ontañón, S. (2013). The combinatorial multi-armed bandit problem and its application to real-time strategy games. In *Proceedings of the AAAI Artificial Intelligence and Interactive Digital Entertainment conference (AIIDE 2013)*.

Ontañón, S. (2016). Informed Monte Carlo Tree Search for real-time strategy games. In *IEEE Conference on Computational Intelligence and Games (CIG 2016)*.

Ontañón, S., & Buro, M. (2015). Adversarial hierarchical-task network planning for complex real-time games. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence*, pp. 1652–1658. AAAI Press.

Ontañón, S., Mishra, K., Sugandh, N., & Ram, A. (2007). Case-based planning and execution for real-time strategy games. In *Case-Based Reasoning Research and Development*, pp. 164–178. Springer.

Ontañón, S., Synnaeve, G., Uriarte, A., Richoux, F., Churchill, D., & 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.

Saffidine, A., Finnsson, H., & Buro, M. (2012). Alpha-beta pruning for games with simultaneous moves. In *Proceedings of the Association for the Advancement of Artificial Intelligence conference (AAAI 2012)*, Toronto, Canada. AAAI Press.

Sharma, M., Holmes, M. P., Santamaría, J. C., Irani, A., Isbell Jr, C. L., & Ram, A. (2007). Transfer learning in real-time strategy games using hybrid CBR/RL. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 2007)*, Vol. 7, pp. 1041–1046.

Shleyfman, A., Komenda, A., & Domshlak, C. (2014). On combinatorial actions and CMABs with linear side information. In *Proceedings of the European Conference on Machine Learning (ECML 2014)*. Springer.

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

Stanescu, M., Barriga, N. A., & Buro, M. (2014). Hierarchical adversarial search applied to real-time strategy games. In *Proceedings of the AAAI Artificial Intelligence and Interactive Digital Entertainment conference (AIIDE 2014)*.

Stanescu, M., Barriga, N. A., Hess, A., & Buro, M. (2016). Evaluating real-time strategy game states using convolutional neural networks. In *Proceedings of the IEEE Conference on Computational Intelligence in Games (CIG 2016)*.

Tolpin, D., & Shimony, S. E. (2012). MCTS based on simple regret. In *Proceedings of the Association for the Advancement of Artificial Intelligence conference (AAAI 2012)*.

Tromp, J., & Farnebäck, G. (2006). Combinatorics of Go. In *International Conference on Computers and Games*, pp. 84–99. Springer.

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

Usunier, N., Synnaeve, G., Lin, Z., & Chintala, S. (2016). Episodic Exploration for Deep Deterministic Policies: An Application to StarCraft Micromanagement Tasks. *ArXiv e-prints*, *arXiv:1609.02993*.

Weber, B. G., & Mateas, M. (2009). Case-based reasoning for build order in real-time strategy games.. In *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2009)*.

Wender, S., & Watson, I. (2012). Applying reinforcement learning to small scale combat in the real-time strategy game StarCraft: Broodwar. In *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, pp. 402–408. IEEE.