The Combinatorial Multi-Armed Bandit Problem and Its Application to Real-Time Strategy Games

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

Abstract

Game tree search in games with large branching factors is a notoriously hard problem. In this paper, we address this problem with a new sampling strategy for Monte Carlo Tree Search (MCTS) algorithms, called *Naïve Sampling*, based on a variant of the Multi-armed Bandit problem called the *Combinatorial Multi-armed Bandit* (CMAB) problem. We present a new MCTS algorithm based on Naïve Sampling called NaïveMCTS, and evaluate it in the context of real-time strategy (RTS) games. Our results show that as the branching factor grows, NaïveMCTS performs significantly better than other algorithms.

Introduction

How to apply game tree search techniques to games with large branching factors is a well-known difficult problem with significant applications to complex planning problems. So far, Monte Carlo Tree Search (MCTS) algorithms (Browne et al. 2012), such as UCT (Kocsis and Szepesvri 2006), are the most successful approaches for this problem. The key to the success of these algorithms is to sample the search space, rather than exploring it systematically. However, algorithms like UCT 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 algorithms like UCT can handle.

This paper focuses on a new sampling strategy to increase the scale of the problems MCTS algorithms can be applied to. UCT, the most popular MCTS algorithm, frames the sampling policy as a Multi-armed Bandit (MAB) problem. In this paper, we will consider domains whose branching factors that are too large for this approach. Instead, we will show that by considering a variant of the MAB problem called the *Combinatorial Multi-armed Bandit* (CMAB), it is possible to handle larger branching factors.

The main contribution of this paper is the formulation of games with combinatorial branching factors as CMABs, and a new sampling strategy for the CMAB problem that we call *Naïve Sampling*. We evaluate NaïveMCTS, the result of using Naïve Sampling in a MCTS framework in multiple scenarios of a RTS game. The results indicate that for scenarios with small branching factors NaïveMCTS performs similar to other algorithms, such as alpha-beta search and UCT. However, as the branching factor grows, the performance of NaïveMCTS gets significantly better than that of the other methods. All the domains in our experiments where deterministic and fully observable.

The remainder of this paper is organized as follows. First, we introduce the challenges posed by RTS games and some background on MCTS. Then we introduce the CMAB problem, and present Naïve Sampling. We then present NaïveMCTS. After that, we present experimental results of our algorithm in an RTS game. The paper concludes with related work, conclusions, and directions for future research.

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 and are real-time. In this context, "real-time" 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 (like in Chess), but can issue actions simultaneously (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. Additionally, some RTS games are also partially observable and non-deterministic, but we will not deal with those two problems in this paper.

While some of these problems have been addressed, like durative actions (Churchill, Saffidine, and Buro 2012) or simultaneous moves (Kovarsky and Buro 2005; Saffidine, Finnsson, and Buro 2012), the branching factor in RTS games is too large for current state-of-the-art techniques. To see why, we should distinguish what we call *unit-actions* (actions that a unit executes) from *player-actions*. A player-action is the set of all unit-actions issued by a given player at a given decision cycle. The number of possible player-actions corresponds to the branching fac-

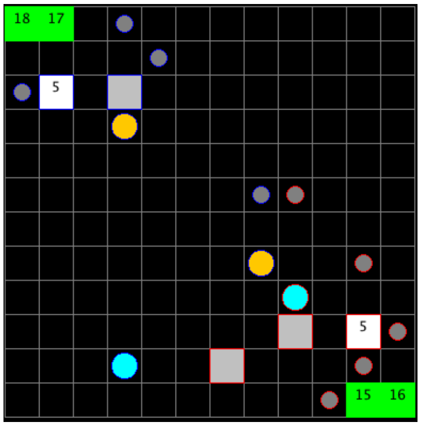

Figure 1: A screenshot of the μRTS simulator.

tor. Thus, the branching factor in a RTS game grows exponentially with the number of units each player controls (since a player can issue actions to an arbitrary subset of units in each decision cycle). As a consequence, existing game tree search algorithms for RTS games resort to using abstraction to simplify the problem (Balla and Fern 2009; Chung, Buro, and Schaeffer 2005).

To illustrate the size of the branching factor in RTS games, consider the situation from the μRTS game (used in our experiments) shown in Figure 1. Two players (blue, on the top-left, and red, on the bottom-right) control 9 units each: the square units correspond to "bases" (that can produce workers), "barracks" (that can produce military units), and "resources mines" (from where workers can extract resources to produce more units), the circular units correspond to workers and military units. 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, or build a barracks or a base in any of the two adjacent cells.

The blue player in Figure 1 can issue 1008288 different player-actions, and the red player can issue 1680550 different player-actions. Thus, even in relatively simple scenarios, the branching factor in these games is very large.

Many ideas have been explored to improve UCT in domains with large branching factors. For example, *first play urgency* (FPU) (Gelly and Wang 2006) allows the bandit policy 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 which of the unexplored nodes to explore first (which is key in our domains of interest). Another idea is to try to better exploit the information obtained from each simulation, like performed by AMAF (Gelly and Silver 2007), however, again, this doesn't solve the problem in the context of RTS games, where the branching factor might be many orders of magnitude larger than the number of simulation we can perform.

Monte Carlo Tree Search

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 s_0. Then, assuming the existence of a reward function R, at each iteration of the algorithm the following three processes are executed:

- SelectAndExpandNode: Starting from the root node of the tree, we choose one of the current node's children according to a *tree policy*, until we reach a node n that was not in the tree before. The new node n is added to the tree.
- Simulation: Then, a Monte Carlo *simulation* is executed starting from n using a *default policy* (e.g. random) to select actions for all the players in the game until a terminal state or a maximum simulation time is reached. Then the reward r obtained at the end of the simulation is returned.
- Backup: Then, 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 time is over, the action that leads to the "best" children of the root node, n_0, is returned. 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 frames the tree policy as a *Multi-arm Bandit* (MAB) problem. MAB problems are a class of sequential decision problems, where at each iteration an agent needs to choose amongst K actions (or 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 $B = \{R_1, ..., R_K\}$, 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.

Solutions to a MAB problem are typically formulated as minimizing the *regret*, i.e. the difference between the obtained accumulated reward and the accumulated reward that would be obtained if we knew beforehand which is the arm with the highest expected reward and always selected it.

UCT uses a specific sampling strategy called UCB1 (Auer, Cesa-Bianchi, and Fischer 2002) that addresses the MAB problem, and 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 and Szepesvri 2006).

Combinatorial Multi-armed Bandits

In this paper, we introduce a variation of the MAB problem, that we call the *Combinatorial Multi-armed Bandit* (CMAB) problem[1]. Specifically, a CMAB is defined by:

[1] The version described in this paper is a generalization a the formulation considered by Gai, Krishnamachari, and Jain (2010).

- A set of n variables $X = \{X_1, ..., X_n\}$, where variable X_i can take K_i different values $\mathcal{X}_i = \{v_i^1, ..., v_i^{K_i}\}$.
- A reward distribution $R : \mathcal{X}_1 \times ... \times \mathcal{X}_n \to \mathbb{R}$ that depends on the value of each of the variables.
- A function $V : \mathcal{X}_1 \times ... \times \mathcal{X}_n \to \{true, false\}$ that determines which variable value combinations are legal.

The problem is to find a legal combination of values of those variables that maximizes the obtained rewards. Assuming that $v_1^*, ..., v_n^*$ are the values for which the expected reward $\mu^* = E(R(v_1^*, ..., v_n^*))$ is maximized, the regret ρ_T of a sampling strategy for a CMAB problem after having executed T iterations is defined as:

$$\rho_T = T\mu^* - \sum_{t=1}^{T} R(x_1^t, ..., x_n^t)$$

where $x_1^t, ..., x_n^t$ are the values selected by the sampling strategy at time t.

Notice that 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 value combination is a different arm. However, in doing so, we would lose the structure (i.e., the fact that each legal value combination is made up of the values of different variables). In some domains, such as RTS games, this internal structure can be exploited, as shown below.

Naïve Sampling for CMAB

Naïve Sampling is a sampling strategy for CMAB, based on decomposing the reward distribution as: $R(x_1, ..., x_n) = \sum_{i=1...n} R_i(x_i)$ (we call this the *naïve assumption*). Thanks to the naïve assumption, we can break the CMAB problem into a collection of $n + 1$ MAB problems:

- MAB_g, that considers the whole CMAB problem as a MAB where each legal variable combination that has been sampled so far is one of the arms. We call this the *global MAB*. Initially, the global MAB contains no arms at all, and in subsequent iterations, all the value combinations that have been sampled, are added to this MAB.
- For each variable $X_i \in X$, we also define a MAB, MAB_i, that only considers X_i. We call these the *local MABs*.

Thus, at each iteration, the global MAB will take into account the following values:

- $T^t(v_1^{k_1}, ..., v_n^{k_n})$ is the number of times that the combination of values $v_1^{k_1}, ..., v_n^{k_n}$ was selected up to time t.
- $\overline{R}^t(v_1^{k_1}, ..., v_n^{k_n})$ is the average reward obtained when selecting values $v_1^{k_1}, ..., v_n^{k_n}$ up to time t.

The local MAB for a given variable X_i will take into account the following values:

- $\overline{R}_i^t(v_i^k)$ is the marginalized average reward obtained when selecting value v_i^k for variable X_i up to time t.
- $T_i^t(v_i^k)$ is the number of times that value v_i^k was selected for variable X_i up to time t.

Intuitively, Naïve Sampling uses the local MABs to *explore* different value combinations that are likely to result on a high reward (via the naïve assumption), and then uses the global MAB to *exploit* the value combinations that obtained the best reward so far. Specifically, the Naïve Sampling strategy works as follows. At each round t:

1. Use a policy π_0 to determine whether to *explore* (via de local MABs) or *exploit* (via the global MAB).
   - If *explore* was selected: $x_1^t, ..., x_n^t$ is sampled by using a policy π_l to select a value for each $X_i \in X$ independently. As a side effect, the resulting value combination is added to the global MAB.
   - If *exploit* was selected: $x_1^t, ..., x_n^t$ is sampled by using a policy π_g to select a value combination using MAB_g.

In our experiments, π_0 was an ϵ-greedy strategy (ϵ probability of selecting *explore* and $1 - \epsilon$ of selecting *exploit*), π_l was also an ϵ-greedy strategy, and π_g was a pure greedy strategy (i.e. an ϵ-greedy with $\epsilon = 0$). However, other MAB policies, such as UCB-based ones can be used.

Intuitively, when exploring, the naïve assumption is used to select values for each variable, assuming that this can be done independently using the estimated $\overline{R}_i^t$ expected rewards. At each iteration, the selected value combination is added to the global MAB, MAB_g. Then, when exploiting, the global MAB is used to sample amongst the explored value combinations, 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 value combinations, and then a regular MAB strategy is used over the global MAB to select the optimal action.

If the policy π_l is selected such that each value 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 $\overline{R}^t$ constantly decreases. As a consequence, the optimal value combination will eventually have the highest estimated reward. This will happen, even for reward functions where the naïve assumption is not satisfied.

For the particular case that all π_0, π_l and π_g are ϵ-greedy policies (with parameters ϵ_0, ϵ_l and ϵ_g respectively), it is easy to see that Naïve Sampling has a linear growth in regret. If the reward function R satisfies the naïve assumption, and π_l is selected such that each value has a non-zero probability, in the limit, the probability of selecting the optimal action in a given iteration is:

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

Where N is the total number of legal value combinations. For example, if $\epsilon_0 = \epsilon_l = \epsilon_g = 0.1$, and we have 10 variables, with 5 values each, then $p \simeq 0.8534$. In case the naïve assumption is not satisfied, then, the only thing we can say is that $p \geq \epsilon_0\epsilon_g$. Thus, the regret grows linearly as: $\rho_T = O(T(1-p)\Delta)$, where Δ is the difference between the expected reward of the optimal action μ^* and the expected average reward of all the other value combinations. Using other policies, such as variable epsilon, or UCB, better, logarithmic, bounds on the regret can be achieved[2]. However, as we will show below, even having a linear growth in regret Naïve Sampling can handle domains with combinatorial branching factors better than other sampling policies.

In order to illustrate the advantage of Naïve Sampling over standard ϵ-greedy or UCB1, let us use the CMAB corresponding to the situation depicted in Figure 1 from the perspective of the blue player (top-left). There are 9 variables (corresponding to the 9 units controlled by the player), and the unit-actions for each of those units are the values that each variable can take. The V function is used to avoid those combinations of actions that are illegal (like sending two units to move to the same cell). Let us now consider three sampling strategies: UCB1, ϵ-greedy with $\epsilon = 0.1$, and Naïve Sampling with $\pi_0 = \pi_l = \pi_g = \epsilon$-greedy with $\epsilon = 0.1$. As the reward function, we will use the result of running a Monte Carlo simulation of the game during 100 cycles (using a random action selection policy), and then using the same evaluation function as used in our experiments (described below) to the resulting game state.

Figure 2 shows the average reward of the player-action considered as the best one so far by each strategy at each point in time (this corresponds to the expected reward of the player-action that would be selected). We run 10000 iterations of each sampling policy, and the plot shows the average of repeating this 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 1008288 legal player-actions. We also tested UCB1, augmented with first play urgency (FPU) (Gelly and Wang 2006), to allow UCB1 to exploit some actions before it has finished exploring all the available actions at least once, however, it still performed much worse than Naïve Sampling. For FPU, we experimented with initial values for unvisited nodes between 4.0 and up to 6.0 in intervals of 0.1 (anything below or above results in a pure random, or pure exploitation), and found 4.9 to be the best.

Intuitively, the main advantage from Naïve Sampling comes from the fact that if a unit-action v for a given unit is found to obtain a high reward in average, then other player-actions that contain such unit-action are likely to be sampled. Thus, it exploits the fact that player-actions with similar unit-actions might have similar expected rewards. We would like to note that there are existing sampling strategies, such as HOO (Bubeck et al. 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.

Naïve Monte Carlo Tree Search in RTS Games

This section presents the NaïveMCTS (Naïve Monte Carlo Tree Search) algorithm, a MCTS algorithm designed for RTS games by exploiting Naïve Sampling.

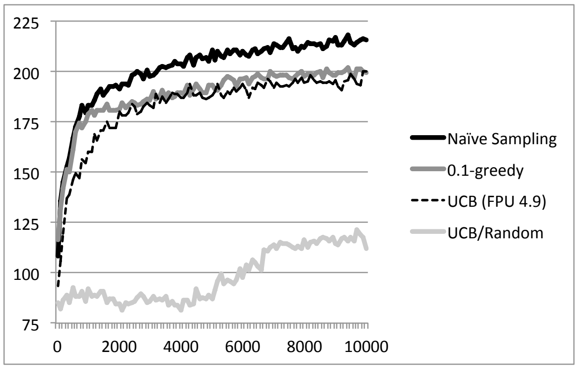

Figure 2: Average expected reward of the best action found so far using four different sampling strategies in a CMAB.

Algorithm 1 SelectAndExpandNode(n_0)

```
if canMove($max, n_0.state$) then
    $player = max$
else
    $player = min$
end if
$a$ = NaïveSampling($n_0.state, player$)
if $a \in n_0.children$ then
    return NaïveSelectAndExpandNode($n_0.child(a)$)
else
    $n_1$ = newTreeNode(fastForward($s_0, a$))
    $n_0$.addChild($n_1, a$)
    return $n_1$
end if
```

Unit-actions issued 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 algorithm (Churchill, Saffidine, and Buro 2012).

Additionally, 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 and Buro 2005; Saffidine, Finnsson, and Buro 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.

NaïveMCTS is designed for deterministic two-player zero sum games, where one player, max, attempts to maximize the reward function R, and the other player, min, attempts to minimize it. NaïveMCTS differs from other MCTS al-

[2] Note, however, that minimizing regret is related, but not equivalent to the problem of finding the best arm.

Table 1: Properties of the different maps used in our experiments: size in cells, maximum number of units observed in our experiments per player, average and maximum branching factor, and average and maximum number of player- and unit-actions that a player executed to win the game.

	Melee2vs3	Melee6vs6	FullGame8x8
Size	4x4	8x8	8x8
Units	2	6	15
Branch.	8.37 / 23	116.07 / 8265	7922.13 / 623700
Actions	12.75 / 18.92	75.17 / 148.00	146.25 / 513.08

gorithms in the way in which the *SelectAndExpandNode* process is defined (which, as explained before, determines which nodes in the game tree are selected to be expanded).

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 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 we recursively apply *SelectAndExpandNode* from that node (i.e. we go down the tree). Otherwise (lines 10-12), a new node is created by executing the effect of player-action a 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.

Therefore, as shown in Algorithm 1, the two key differences of NaïveMCTS with respect to other MCTS algorithms is the use of Naïve Sampling, and accounting for durative actions (through the *fastForward* function, and by not assuming that players alternate in executing actions).

Experimental Results

In order to evaluate the performance of NaïveMCTS, we used the open-source μRTS[3]. We ran experiments in different two-player μRTS maps, as shown in Table 1: two melee maps (with only military units in the map), and one standard game map (where each player starts with one base and one worker). As we can see, the selected domains vary in complexity, *Melee2vs2* is the simplest, with a maximum branching factor of 24, and requiring an average 12.75 player-actions to complete a game. *FullGame8x8* is the most complex, with branching factors reaching 623700.

In our experiments, we used the following AIs:

- *RandomBiased*: selects one of the possible player-actions at random, but with 5 times more probability of selecting an attack or a harvest action than any other action.
- *LightRush*: A hard-coded strategy. Builds a barracks, and then constantly produces ”Light” military units to attack the nearest target (it uses one worker to mine resources).
- *ABCD*: The ABCD algorithm (Churchill, Saffidine, and Buro 2012), an alpha-beta algorithm that takes into account durative actions, implements a tree alteration technique to deal with simultaneous actions, and uses a playout policy. We used a *WorkerRush* strategy, producing workers rather than military units, as the playout policy (which obtained the best results in our experiments).
- *Monte Carlo*: A standard Monte Carlo search algorithm: for each legal player-action, it runs as many simulations as possible to estimate their expected reward.
- *ϵ-Greedy Monte Carlo*: Monte Carlo search, but using an ϵ-greedy sampling strategy (we tested $\epsilon \in \{0.1, 0.15, 0.2, 0.25, 0.33\}$ and chose 0.25 as the best).
- *Naïve Monte Carlo*: Standard Monte Carlo search, but using Naïve Sampling. We used ϵ-greedy policies for π_0, π_l and π_g, with $\epsilon_0 = 0.75$, $\epsilon_g = 0$, and $\epsilon_l = 0.33$ respectively, (selected experimentally, after evaluating all the combinations of $\epsilon_0 \in \{0.25, 0.33, 0.5, 0.75\}$, $\epsilon_g \in \{0.0, 0.1, 0.25, 0.33\}$, and $\epsilon_l \in \{0.0, 0.1, 0.25, 0.33\}$).
- *UCT*: standard UCT, using a UCB1 sampling policy.
- *ϵ-Greedy MCTS*: Like NaïveMCTS, but using an ϵ-greedy sampling strategy ($\epsilon = 0.25$) instead of Naïve Sampling.
- *NaïveMCTS*: we used ϵ-greedy policies for π_0, π_l and π_g, with $\epsilon_0 = 0.75$, $\epsilon_g = 0$, and $\epsilon_l = 0.33$ respectively, selected experimentally.

All the AIs that required a policy for Monte Carlo simulations used the *RandomBiased* AI limited to simulating 100 game cycles (except ABCD, which works better with a deterministic policy). Also, all the AIs that required an evaluation function used the following one: sum the cost in resources of all the player units in the board weighted by the square root of the fraction of hit-points left, then subtract the same sum for the opponent player.

For each pair of AIs, we ran 20 games per map. We limited each game to 3000 cycles (5 minutes), after which we considered the game a tie. Experiments were run in an Intel Core i5-2400 machine at 3.1GHz, on which our implementation of the Monte Carlo-based algorithms had time to run an average of 26326.38, 8871.08 and 5508.99 simulations per decision in each of the maps respectively (simulations in more complex maps required more CPU time).

Figure 3 shows the summarized results of our experiments. For each scenario and for each AI, we show a “score”, calculated as $wins + 0.5ties$. We can clearly observe that in the simple *Melee2vs2* scenario, all AIs perform almost identical (except for *RandomBiased*). In the more complex *Melee6vs6*, we see that ABCD can still defeat hard-coded strategies like *LightRush*, but cannot compete with Monte Carlo-based approaches. UCT still performs well in this scenario, but not as well as NaïveMCTS. Finally, in the more complex *FullGame8x8* map, only the ϵ-greedy and Naïve Sampling-based approaches performed well (the UCB1 sampling strategy of UCT is not suited for such large branching factors). Again NaïveMCTS was the best performing AI overall.

Finally, Table 2 shows a detailed account of the results we obtained in the *FullGame8x8* scenario. We can see, for

[3] https://code.google.com/p/microrts/

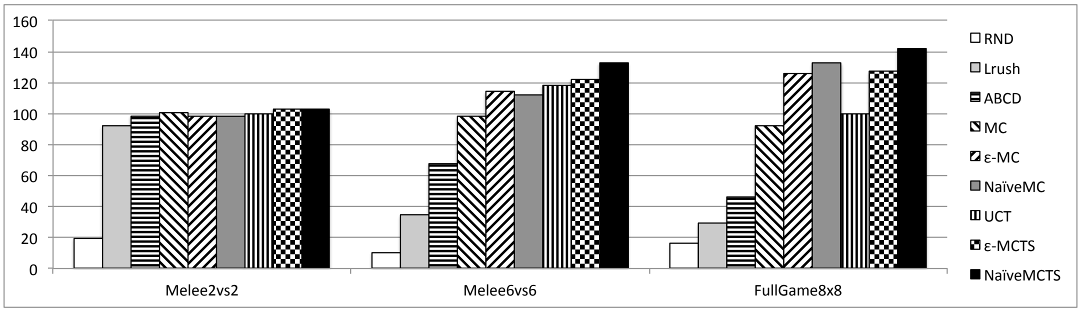

Figure 3: Accumulated score obtained by each AI in each of the different maps: wins plus 0.5 times the number of ties.

Table 2: Wins/ties/loses of the column AI against the row AI in the *FullGame8x8* map.

	RND	LRush	ABCD	MC	ϵ-MC	NaïveMC	UCT	ϵ-MCTS	NaïveMCTS
RND	9/2/9	14/0/6	19/1/0	20/0/0	20/0/0	20/0/0	20/0/0	20/0/0	20/0/0
LRush	6/0/14	0/20/0	10/10/0	20/0/0	20/0/0	20/0/0	20/0/0	20/0/0	20/0/0
ABCD	0/1/19	0/10/10	3/14/3	19/1/0	20/0/0	20/0/0	18/2/0	20/0/0	20/0/0
MC	0/0/20	0/0/20	0/1/19	8/4/8	14/1/5	17/1/2	10/2/8	15/1/4	19/0/1
ϵ-MC	0/0/20	0/0/20	0/0/20	5/1/14	10/0/10	10/2/8	4/0/16	12/2/6	11/0/9
NaïveMC	0/0/20	0/0/20	0/0/20	2/1/17	8/2/10	10/0/10	4/0/16	9/0/11	13/0/7
UCT	0/0/20	0/0/20	0/2/18	8/2/10	16/0/4	16/0/4	8/4/8	11/2/7	16/1/3
ϵ-MCTS	0/0/20	0/0/20	0/0/20	4/1/15	6/2/12	11/0/9	7/2/11	9/2/9	10/5/5
NaïveMCTS	0/0/20	0/0/20	0/0/20	1/0/19	9/0/11	7/0/13	3/1/16	5/5/10	8/4/8
Total	15/3/162	14/30/136	32/28/120	87/10/83	123/5/52	131/3/46	94/11/75	121/12/47	137/10/33

example, that NaïveMCTS defeated ϵ-MCTS 10 times, losing only 5 times, and that it defeated UCT 16 times, losing only 3 times. We evaluated many other AIs, such as randomized alpha-beta (Kovarsky and Buro 2005) (which performed worse than ABCD), two other hard-coded AIs and many different parameter settings of the MCTS strategies, but lack of space prevents us from showing their results.

Related Work

Concerning the application of Monte Carlo algorithms to RTS games, Chung et al. (2005) proposed the MCPlan algorithm. MCPlan uses *high-level plans*, where a plan consists of a collection of destinations for each of the units controlled by the AI. At the end of each simulation, an evaluation function is used, and the plan that performed better overall is selected. The idea was continued by Sailer et al. (Sailer, Buro, and Lanctot 2007) where they studied the application of game theory concepts to MCPlan.

A more closely related work to NaïveMCTS is that of Balla and Fern (2009), who study the application of UCT (Kocsis and Szepesvri 2006) to the particular problem of tactical battles in RTS games. In their work, they use abstract actions that cause groups of units to merge or attack different enemy groups. Another application of UCT to real-time games is that of Samothrakis et al. (2011), in the game Ms. Pac-Man, where they first re-represent Ms. Pac-Man as a turn-based game, and then apply UCT.

Many other approaches have been explored to deal with RTS games, such as case-based reasoning (Ontañón et al. 2010; Aha, Molineaux, and Ponsen 2005) or reinforcement learning (Jaidee and Muñoz-Avila 2012). A common approach is to decompose the problem into smaller subproblems (scouting, micro-management, resource gathering, etc.) and solving each one individually, as done in most bots in the StarCraft AI competition (Uriarte and Ontañón 2012; Churchill and Buro 2011; Synnaeve and Bessiere 2011; Weber, Mateas, and Jhala 2011).

Conclusions

This paper has presented NaïveMCTS, a Monte Carlo Tree Search algorithm designed for games with a combinatorial branching factor, such as RTS games, where the magnitude of the branching factor comes from the fact that multiple units can be issued actions simultaneously. At the core of NaïveMCTS, is Naïve Sampling, a strategy to address the Combinatorial Multi-armed Bandit problem.

Experimental results indicate that NaïveMCTS performs similar to other algorithms, like ABCD or UCT, in scenarios with small branching factors. However, as the branching factor grows, NaïveMCTS gains a significant advantage. The main reason for this is that Naïve Sampling can guide the exploration of the space of possible player-actions, narrowing down the search on the most promising ones.

As part of our future work, we plan to explore the performance of NaïveMCTS in even larger scenarios (we are currently working on applying it to the commercial RTS game *Starcraft*), which will require the use of abstraction. We would also like to design better sampling strategies for the CMAB problem, and evaluate their performance versus Naïve Sampling in the context of RTS games.

References

Aha, D.; Molineaux, M.; and Ponsen, M. 2005. Learning to win: Case-based plan selection in a real-time strategy game. In *ICCBR'2005*, number 3620 in LNCS, 5–20. Springer-Verlag.

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

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

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

Bubeck, S.; Munos, R.; Stoltz, G.; Szepesvari, C.; et al. 2008. Online optimization in 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 IJCAI 2003*, 1534–1535. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Chung, M.; Buro, M.; and Schaeffer, J. 2005. Monte carlo planning in rts games. In *Proceedings of IEEE-CIG 2005*.

Churchill, D., and Buro, M. 2011. Build order optimization in starcraft. *Proceedings of AIIDE* 14–19.

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

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

Gelly, S., and Silver, D. 2007. Combining online and offline knowledge in uct. In *Proceedings of the 24th international conference on Machine learning*, 273–280. ACM.

Gelly, S., and Wang, Y. 2006. Exploration exploitation in go: Uct for monte-carlo go.

Jaidee, U., and 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*.

Kocsis, L., and Szepesvri, C. 2006. Bandit based monte-carlo planning. In *Proceedings of ECML 2006*, 282–293. Springer.

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

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

Saffidine, A.; Finnsson, H.; and Buro, M. 2012. Alpha-beta pruning for games with simultaneous moves. In *26th AAAI Conference (AAAI)*. Toronto, Canada: AAAI Press.

Sailer, F.; Buro, M.; and Lanctot, M. 2007. Adversarial planning through strategy simulation. In *Proceedings of IEEE-CIG 2007*, 80–87.

Samothrakis, S.; Robles, D.; and Lucas, S. M. 2011. Fast approximate max-n monte carlo tree search for Ms Pac-Man. *IEEE Trans. CI and AI in Games* 3(2):142–154.

Synnaeve, G., and Bessiere, P. 2011. A Bayesian Model for RTS Units Control applied to StarCraft. In *Proceedings of IEEE CIG 2011*, 000.

Uriarte, A., and Ontañón, S. 2012. Kiting in rts games using influence maps. In *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*.

Weber, B. G.; Mateas, M.; and Jhala, A. 2011. A particle model for state estimation in real-time strategy games. In *Proceedings of AIIDE*, 103–108. Stanford, Palo Alto, California: AAAI Press.