Proceedings of the Fifteenth AAAI Conference on Artificial
Intelligence and Interactive Digital Entertainment (AIIDE-19)

Guiding Monte Carlo Tree Search by Scripts in Real-Time Strategy Games

Zuozhi Yang, Santiago Ontañón
Drexel University, Philadelphia, Pennsylvania 19104
{zy337, so367}@drexel.edu

Abstract

In Real-Time Strategy (RTS) games, the action space grows combinatorially with respect to the number of units. With limited computing budget between actions, methods like Monte Carlo Tree Search (MCTS) tend to get lost in the massive search space. An interesting line of existing work is to incorporate human knowledge in the form of scripts. In this paper, we investigate different possibilities for incorporating scripts into the tree policy while still maintaining the convergence guarantees of MCTS. We also report experiments on incorporating the scripts into the playout policy, which showed that unbiased bots perform better than biased bots.

Introduction

Real-Time Strategy (RTS) games offer a rich testbed for artificial intelligence algorithms for researchers for their complexity and real-time nature. Building an RTS game playing agent is considered a challenging problem that is currently being studied both in the machine learning and game tree search AI communities. Despite recent advances in building strong agents using deep learning and reinforcement learning (e.g. *AlphaStar*), there are still many open problems such as robustness or long term planning, which can potentially be addressed by search techniques.

Monte Carlo Tree Search (MCTS) (Browne et al. 2012; Coulom 2007) is a well known game tree search algorithm that was initially intended to address the large branching factor and lack of good evaluation function in the game of Go. The large branching factor problem is even more pronounced in RTS games, especially since the amount of time to make a decision is more limited than in games like Go. Thus improving the scalability of MCTS is one of the key problems in deploying search-based techniques to RTS games. While previous work has shown that machine-learned stochastic policies can be used to improve the performance of MCTS (Silver et al. 2016; Ontañón 2016), in this paper, we study how to integrate human knowledge, in the form of hard-coded (deterministic) scripted bots, into MCTS in order to improve the gameplay strength of MCTS under tight computational budget constraints. Incorporating scripts into MCTS algorithms is not a new idea (Justesen et al. 2014; Barriga, Stanescu, and Buro 2017; Moraes and Lelis 2017), but the key contribution of this paper is to present methods to integrate scripts into MCTS without pruning the search space, and thus, still ensuring that MCTS would converge to the global optimum in the limit. Our experiments in μRTS (Ontañón 2017) show win rates up to 94% against the baseline MCTS implementation.

The rest of the paper is structured as follows: first we first talk about related work on applying MCTS to RTS games, and then we describe the research environment we use. After that, we describe two new approaches to modify MCTS to incorporate scripts. Then we describe and report our experimental setup and results. Finally, we talk about conclusions and possible future work.

Background

In RTS games, players control a large number of units to mine resources, build more units, and combat with other players. RTS games have been receiving an increased amount of attention as they are even more challenging than games like Go or Chess in at least three different ways: (1) the combinatorial growth of the branching factor (Ontañón 2017), (2) limited computation budget between actions due to the real-time nature, and (3) lack of forward model in most of research environments like StarCraft. Many research environments and tools, such as TorchCraft (Synnaeve et al. 2016), SCIILE (Vinyals et al. 2017), μRTS (Ontañón 2017), ELF (Tian et al. 2017), and Deep RTS (Andersen, Goodwin, and Granmo 2018) have been developed to promote research in the area. Specifically, in this paper, we chose μRTS as our experimental domain, as it offers a forward model for game tree search approaches such as minimax or MCTS.

The number of possible actions a player can execute in a given game state in an RTS game grows combinatorially with respect to the number of units the player controls. This enormous action space poses a great challenge to both machine learning approaches and game tree search approaches to build strong RTS game agents. To alleviate this problem, recent work in reinforcement learning on RTS games (Sun et al. 2018) limit the action space by defining a set of macro actions using domain knowledge.

Monte Carlo Tree Search in RTS Games

Monte Carlo Tree Search (Browne et al. 2012; Coulom 2007) is a method for sequential decision making for domains that can be represented by search trees. It has been a successful approach to tackle complex games like Go as it takes random samples in the search space to estimate state value rather than systematically exploring the whole tree. However, most of the successful variants of MCTS, e.g. UCT (Kocsis and Szepesvári 2006), do not scale up well to RTS games due to the combinatorial growth of branching factor with respect to the number of units. Sampling techniques for combinatorial branching factors such as Naïve Sampling (Ontañón 2017) or LSI (Shleyfman, Komenda, and Domshlak 2014) were proposed to improve the exploration of MCTS exploiting combinatorial multi-armed bandits. Inspired by AlphaGo, another approach to address this problem is the use of learned probabilistic models to be used as a prior to guide the search (Ontañón 2016). Other work to deal with this problem involves limiting the search space by introducing action abstractions build from human-authored scripts. For example, instead of searching directly in the raw unit action space, Portfolio Greedy Search (Churchill and Buro 2013), Stratified Alpha-Beta Search (Moraes and Lelis 2017) search in the abstracted action spaces generated by hard-coded scripts, and techniques like asymmetric action abstractions prune the search space by only considering the full action space for a subset of units in the game, and using scripts to control the rest (Moraes et al. 2018).

Another related idea is that of using previously trained policies to guide the search (e.g., as used in AlphaGo (Silver et al. 2016)). An example multiarmed bandit strategy that incorporates this is PUCB (Predictor + UCB) (Rosin 2011), which modifies the standard UCB1 policy to incorporate an action probability distribution provided by the predictor. Moreover, these approaches assume the external policy can provide a probability distribution of actions. In this work, we are interested in incorporating simple (potentially deterministic) scripts that are simpler to specify and available for many RTS game domains in a way that search can greatly benefit from them, and without modifying the convergence guarantees of MCTS.

Naïve Monte Carlo Tree Search NaïveMCTS (Ontañón 2017) is a variant of MCTS specifically designed to handle RTS games. NaïveMCTS can handle durative and simultaneous actions, but most importantly, the key feature of NaïveMCTS is that rather than using standard ϵ-greedy or UCB1, it uses a sampling policy based on Combinatorial Multiarmed Bandits (CMABs) called *Naïve Sampling* in order to scale up to the combinatorial branching factors of RTS games. All the experiments reported in this paper use NaïveMCTS as the baseline, thus, we briefly describe below.

MCTS algorithms employ two policies called the *tree* policy and the *default* or *playout* policy. The former is used to determine which node of the tree to explore next, and the later is used to perform stochastic rollouts from the leaf of the tree until a terminal state is reached.

Naïve Sampling is a sampling strategy for CMAB problems designed to act as the tree policy of MCTS. Thus, it tries to find the *macro-action* (the combination of all the actions issued to game units in a given game state by the player) that maximizes the expected reward. As usual, Naïve Sampling decomposes this problem into *exploration* and *exploitation*. During exploration, the *naïve assumption* that the reward of the macro-action can be decomposed as the sum of the expected rewards of the individual unit actions is used. With this assumption, we can decompose the CMAB problem into n child MAB problem, denoted as $MAB_1, \ldots, MAB_n$. During *exploitation*, the *naïve assumption* is ignored, and a global MAB, denoted as MAB_g (which has a macro-arm for each macro-action generated so far during exploration), is used to find the best macro-action amongst all the currently explored ones, ensuring convergence to the optimal macro-action regardless of whether the domain at hand satisfies or not the naïve assumption.

Specifically Naïve Sampling works as follows. At each iteration, a stochastic decision is made to exploit (with probability $1 - \epsilon_0$) or explore (with probability ϵ_0):

- if exploit is selected: an ϵ-greedy policy π_g is used to select a macro-action using the global MAB.
- if explore is selected: for each local MAB, an ϵ-greedy policy π_l is used independently for each game unit to select actions, and the resulting macro-action is added to the global MAB.

μRTS

μRTS[1] is a simple RTS game maintaining the essential features that make RTS games challenging from an AI point of view: simultaneous and durative actions, combinatorial branching factors and real-time decision making. Although the game can be configured to be partially observably and non-deterministic, but those settings are turned off for all the experiments presented in this paper. Additionally, μRTS allows maps of arbitrary sizes and initial configurations, which we exploit in our experimental evaluation.

There is one type of environment unit (minerals) and six types of units controlled by players, which are:

- *Base*: can train Workers and accumulate resources
- *Barracks*: can train attack units
- *Worker*: collects resources and construct buildings
- *Light*: low power but fast melee unit
- *Heavy*: high power but slow melee unit
- *Ranged*: long range attack unit

Additionally, the environment can have walls to block the movement of units. A screenshot of game is shown in Figure 1. The squared units in green are Minerals with numbers on them indicating the remaining resources. The units with blue outline belong to player 1 (which we will call *max*) and those with red outline belong to player 2 (which we will call *min*). The light grey squared units are Bases with numbers indicating the amount of resources owned by the player, while the darker grey squared units are the Barracks. Movable units have round shapes with grey units being Workers,

[1] https://github.com/santiontanon/microrts

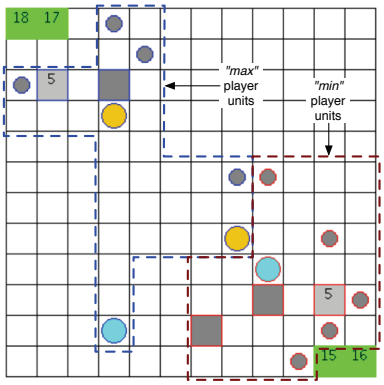


Figure 1: A screenshot of μRTS, highlighting the units controlled by each of the two players.

orange units being Lights, yellow being Heavy units (not shown in the figure) and blue units being Ranged.

Guided Naïve Monte Carlo Tree Search

Due to the limited search budget in RTS games and massive branching factor, it is usually unfeasible to produce a reliable estimation of the expected reward of the possible actions from sampling. Meanwhile, scripted bots are hard-coded agents using human knowledge, which follow simple and fast strategies, sometimes achieving a good level of gameplay strength. For example, scripted bots performed among the best bots in the first μRTS AI competition (Ontañón et al. 2018). In this paper, we propose two tree policies to incorporate scripted bots (which we will just call "scripts") as guidance for game tree search. A key characteristic of our approach, however, is that we use those scripts only to guide the exploration of the tree in MCTS, instead of for pruning the search space, as had been done in the past. In this way, we preserve the original search space of MCTS unchanged, and under certain conditions (satisfied in our experiments), the resulting MCTS algorithms will still converge to the optimal action in the limit (while previous approaches based on scripts might accidentally prune the optimal action of the search space, and must settle for just finding the best action among those included in the reduced search space defined by the scripts).

First-Choice Guided Naïve Sampling (FC-GNS)

This tree policy utilizes a single script to inform the search. The first time the tree policy is used on a search node of the MCTS treee to determine which is the first child that will be added to such node, instead of using stochastic sampling (like Naïve Sampling or ϵ-greedy do), or choosing the first non-expanded child (as UCB1 does), we propose to choose the macro-action suggested by the script. In all other circumstances, the tree policy selects a player action that consists of unit actions selected by Naïve Sampling. The resulting tree policy is shown in Algorithm 1.

The key idea behind this is that while the top nodes of the tree might be explored heavily by MCTS, as we go deeper into the tree, most nodes would only be visited once or a handful of times. Thus, by first selecting an action based on a script, we ensure that if a node is only visited once, at least the action selected is an action that makes sense, and not a random action. Moreover, since the script is only used the first time a node is visited, in the long-run, MCTS can ignore it if other actions turn out to have higher reward. Thus, this does not change the theoretical convergence guarantees of MCTS, and would still find the optimal action in the limit.

Algorithm 1: FC-GNS Node Expansion(n_0, s)

if $n_0.children = \emptyset$ **then**
  $n_1 = n_0.newNode(s.getAction())$;
else
  $n_1 = n_0.newNode(NaiveSampling(n_0))$;
end

Mixed Scripts Guided Naïve Sampling (*k*-GNS)

This tree policy utilizes a collection of k scripts ($k \geq 1$) to guide the search. Scripts are used in two different ways:

- In the very first iteration in each node of the search tree: similarly to FC-GNS, the first time the tree policy is used, the scripts are used to chose the selected action. Rather than using a single script, k-GNS chooses a script uniformly from a pool of scripts, and uses the action produced by that script.
- In each subsequent iteration: when using a local MABs in Naïve Sampling, with probability ϵ, the scripts will be used to select the unit action, and with probability $1 - \epsilon$, the local MABs will be used. Thus, notice that this means that, in average an ϵ proportion of the units would have their actions assigned by scripts when using the local MABs in Naïve Sampling.

In our experiments, unless otherwise noted, we use $\epsilon = 0.33$. Larger ϵ values force the search towards the scripts, which can be useful in larger maps as we will show later in the experiments. Moreover, as long as $\epsilon < 1$ (so that all possible macro actions have a non zero probability of being chosen at some point), and $\epsilon_0 > 0.5$ in Naïve Sampling (so that exploitation can overpower the action selection imposed by the scripts) we can ensure that MCTS can still converge to the optimal action in the limit.

The resulting tree policy is shown in Algorithm 2. The idea behind having more than one script, and inserting this probability ϵ of using the scripts at any iteration is to bring the actions proposed by the scripts to the attention of MCTS more often, and thus more strongly guide the search process. As we will show in our experimental results, this has positive effects, specially for larger maps where the baseline NaïveMCTS gets lost in the search space. Moreover, as we will also see, k-GNS is stronger than both NaïveMCTS and than the individual scripts used to guide the search, as NaïveMCTS can some times find actions that are better than those proposed by the scripts (but when it can't, it can at least rely on the scripts).

Finally, in our experiments, we will use the name 1-GNS to refer to the degenerate form of k-GNS when $k = 1$.

Algorithm 2: k-GNS Node Expansion(n_0, S, ϵ)

if $n_0.children = \emptyset$ *or with probability* ϵ **then**
  s = randomly select from S
  $n_1 = n_0.newNode(s.getAction())$;
else
  $n_1 = n_0.newNode(NaiveSampling(n_0, S))$;
end

Experiments

In order to evaluate our approaches, we performed experiments in μRTS, and used maps of five different sizes: 8×8, 12×12, 16×16, 32×32, and 128×128. For each map, both players always start with one base and one worker. Since FC-GNS and k-GNS are basically the result of integrating NaïveMCTS with scripts, we compare the performance of our algorithms against both NaïveMCTS and the scripts used for guidance (ideally, the combined approach should be better than both NaïveMCTS and the scripts).

For experiments comparing FC-GNS/k-GNS against NaïveMCTS, we used 8×8, 12×12, and 16×16 maps, as NaïveMCTS does not stand a chance to win a single game on maps larger than that. For experiments comparing our algorithms against the individual scripts used for guidance, we used all five map sizes. For each map, 100 games are played between guided NaïveMCTS and the baselines. Thus, each variation of guided NaïveMCTS is tested in 100 games per map size. Games that go beyond 5000 cycles are considered a draw and in each cycle we give players a computation budget of 500 iterations of MCTS, except in the 128×128 map, where a smaller budget of 100 iterations is given due to computational cost (running experiments in such a large map is much slower than on the smaller maps). Moreover, notice that this would, if anything, handicap our algorithms, rather than benefit them, as the opponents are scripts. We report the 95% confidence interval for all win percentage results.

Scripted Bots used for Guidance

We used four simple scripted bots provided by μRTS as guidance bots:

- *WorkerRush*: hardcoded deterministic bot that constantly produces Worker units and sends them to attack the opponent's units and base
- *LightRush*: hardcoded deterministic bot that constantly produces Light units and sends them to attack the opponent's units and base
- *HeavyRush*: hardcoded deterministic bot that constantly produces Heavy units and sends them to attack the opponent's units and base
- *RangedRush*: hardcoded deterministic bot that constantly produces Ranged units and sends them to attack the opponent's units and base

Experiments with FC-GNS

We run experiments of FC-GNS with four different scripts playing against vanilla NaïveMCTS in three set of maps with increasing sizes. The aggregated win ratio results are shown in Table 1 where 1.0 means FC-GNS wins 100% of the time, and 0.0 would mean NaïveMCTS wins 100% of the times. Results show that the performance of this approach gets better as the map size grows. The reason is that the action space grows with the map size and makes it even harder for the search to get accurate estimations, thus the script helps in the search process. In smaller maps MCTS search alone can find good actions by itself and the improvement is not significant. We also see that some scripts are better than others at providing guidance, and that LightRush seems to obtain the best results, achieving a 76.88% win ratio in 16×16 maps.

Table 1: Win Rates of FC-GNS guided by different scripts against NaïveMCTS.

	8×8	12×12	16×16
WorkerR	0.531±0.062	0.652±0.060	0.619±0.061
LightR	0.571±0.062	0.644±0.060	0.769±0.053
HeavyR	0.540±0.062	0.613±0.061	0.706±0.057
RangedR	0.504±0.063	0.571±0.063	0.719±0.056

Figure 2 contains the results of FC-GNS (grey) and NaïveMCTS (yellow) versus the scripts. We can see that in all sizes of maps, FC-GNS achieved better performance than the baseline NaïveMCTS. We also see that FC-GNS has a higher than 50% win rate against the scripts in 8×8, 12×12 and 32×32, but cannot defeat the scripts in the 16×16 and the 128×128 maps Especially in the 128×128, FC-GNS hardly win against the script, suggesting the guidance is insufficient for such huge search space. In the next section, we show that k-GNS can mitigate the problem by incorporating more and varied guidance.

Experiments with *k*-GNS

The combination of multiple scripts provided diversified yet good directions to explore. The results shown in Table 2 continue the positive trend shown in the results of FC-GNS, but we see even larger gains. Specifically, we see that our agents do no better than vanilla NaïveMCTS in 8×8 maps but has better performance in 12×12 and 16×16 maps (win rate of 95.83%, which is remarkable).

Additionally, we also compared the performance of FC-GNS, 1-GNS and k-GNS against the scripts in all five maps for reference. The ϵ we used in the first four map sizes is 0.33. For the experiments in 128×128 maps we used 0.75, as we saw it worked better in initial experiments (with 0.33, we still observed our approach significantly outperforming the scripts, but there were many draws, as guidance wasn't strong enough). Exploring the tradeoff between the amount of guidance and the performance in different map sizes/types is still part of our future work. For the NaïveMCTS and k-GNS, we run experiments against each of the rush scripts (and present the average), and for FC-GNS and 1-GNS, we run experiments against the corresponding guiding script. The aggregated results are shown in Figure 2. We found that FC-GNS improved upon NaïveMCTS on their performance against the scripts in all maps. And 1-GNS and k-GNS had

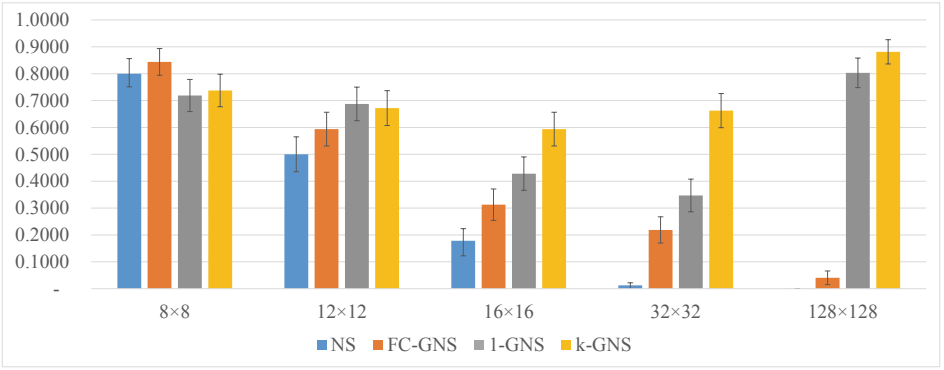
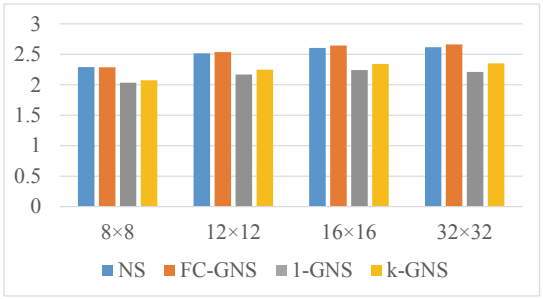
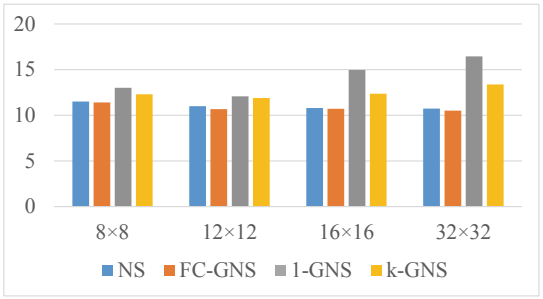


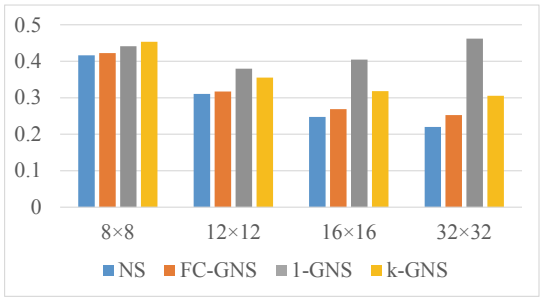
Figure 2: Win rates of NaïveMCTS, FC-GNS and k-GNS against the four rush scripts



(a) Average Number of Children per Node



(b) Max Tree Depth



(c) Proportion the best action was visited at the top level

Figure 3: Comparison of Search Tree Statistics between Tree Policies with and without Guidance

even larger improvements except for the 8×8 map. This is because in very small maps, the search algorithm is able to find better solution than the scripts and adding the bias of the scripts to the search just hurts performance. It is worth noting that the superiority of 1-GNS and k-GNS really starts to show in larger maps like 32×32 and 128×128. While in maps as large as 128×128, a larger ϵ will help the search get focused in huge search space. In particular, on these large map sizes, we obtained win ratios close to 90% against the scripts with k-GNS (which is remarkable, since in this map size NaïveMCTS does not stand a chance against either the scripts or k-GNS).

Table 2: Win Rates of k-GNS against NaïveMCTS.

	8×8	12×12	16×16
Win %	0.481±0.063	0.790±0.051	0.958±0.024

Looking at some of the games, we saw that in the larger maps, when units take a long time to reach the enemy base, 1-GNS and k-GNS had time to build a strong defense before the scripts came in for the attack, and since 1-GNS and k-GNS harvest resources much faster than the scripts thanks to tree search, they were able to outperform the scripts easily.

In order to understand where do the benefits in performance of FC-GNS and k-GNS come from, we analyze the shape of the resulting search trees below.

Comparison of the Shape of the Search Tree

To further understand the behavior of the proposed algorithms, we collected statistics of the shape of the search trees. The statistics are collected from 48 games between FC-GNS and k-GNS and vanilla NaïveMCTS (24 games for FC-GNS and 24 for k-GNS). We use data from the first 1000 game cycles of each game.

In Figure 3, we compare the shape of the trees generated by FC-GNS and k-GNS to the shape of vanilla NaïveMCTS from three perspectives: (a) average number of children per node, (b) maximum tree depth, and (c) the proportion of times that MCTS visited the selected action at the root of the tree. We can observe that the difference of average number of children per node is insignificant between FC-GNS and the baseline. And the search tree of FC-GNS is slightly deeper than the baseline. While for k-GNS, due to more enforcement on the scripts, the average number of children per node is smaller and the tree is deeper. This could be because

Table 3: Winrate of k-GNS against FC-GNS

	8×8	12×12	16×16
k-GNS	0.283±0.057	0.658±0.060	0.842±0.046

Table 4: Win Rates of FC-GNS with Scripted Playout against NaïveMCTS

	8×8	12×12	16×16
WorkerR	0.388±0.061	0.227±0.052	0.071±0.031
LightR	0.494±0.060	0.279±0.052	0.510±0.062
HeavyR	0.521±0.059	0.119±0.040	0.323±0.058
RangedR	0.529±0.059	0.135±0.041	0.417±0.062

in FC-GNS, the node is only guided by the script at creation, and it does the same thing as the baseline the rest of the time. For k-GNS, the scripted action can be reinforced multiple times, thus the search is more concentrated. Finally, looking at the proportion of time that the selected action was visited at the root of the tree (the larger, the more concentrated the search is in one action), we see that in 8×8 maps, it seems that the scripts tended to distract MCTS rather than guide it, as adding more guidance made the search being less concentrated (which means that the action proposed by the scripts was often not the one selected at the end). However, in the larger maps, the more guidance, the higher the ratio of playouts that went through the selected action, which indicates that the scripts were helping MCTS in focusing the search.

Table 3 shows the proportion of times MCTS selected the action proposed by one of the scripts as the selected action. As can be seen, k-GNS selects actions from the scripts a larger percentage of times (up to 51.28% in 12×12 maps, where as FC-GNS only selects the action proposed by the script up to 18.29% of the times). We also see that in the smaller 8×8 map, MCTS overpowered the scripts more often in k-GNS than for the larger maps.

In summary, FC-GNS generates trees that are very similar to vanilla NaïveMCTS, but k-GNS generates narrower and deeper trees. Moreover, the guidance has a stronger influence in larger maps, where we can see that the percentage of times the final action is one of the actions proposed by the script is higher than the percentage of times vanilla NaïveMCTS would have selected one of those actions.

NaïveMCTS with Guided Playout Policy

Finally, we explored the idea of using the scripts as the playout policy. In most MCTS variants, the playout policy usually just selects random actions until reaching the end of the game, or a pre-defined depth. Due to the limited computing budget in RTS games, it would be beneficial to have a strong playout policy that can provide a good evaluation of the current state. However, as (Silver and Tesauro 2009) pointed out, a stronger policy, whether its learned or hard-coded, does not necessarily lead to stronger gameplay because of the bias in the playout policy.

In all the experiments presented above, the playout policy we used is the *RandomBiased* policy provided by μRTS (which selects actions randomly, but if an attack, harvest or return action is available, it chooses it with 4 times more probability), which, despite its name, is less biased than the rush policies as a playout policy. In Table 4 we report the performance of FC-GNS with each of the corresponding guiding scripts as the playout policies. We can see the result is quite catastrophic when the guiding scripts are used as playout policies. Our hypothesis of the results is that despite the scripts achieving good strength in gameplay, they are still not optimal, and the proposed algorithms work under the premise that the MCTS will refine the player action provided by the scripts. However, applying the scripts as playout policy can prevent the MCTS from recovering from these suboptimalities, since it will assume players will behave as the script indicates. Also, the following situation could happen: 1) when the playout indicates the agent has the edge, the agent plays boldly as the script suggests, however 2) when the playout indicates otherwise, the player action from the scripts gets overridden. This could create a "zigzag" effect, where the agent plays actions that cancels each other out, since in RTS games, macro goals, like "harvest" or "attack", take a series of actions to accomplish.

Table 5: Proportion of a script action being selected.

	FC-GNS	1-GNS	k-GNS
8×8	0.149±0.044	0.391±0.061	0.442±0.062
12×12	0.114±0.040	0.488±0.062	0.493±0.063
16×16	0.143±0.042	0.547±0.061	0.529±0.062
32×32	0.144±0.043	0.582±0.061	0.517±0.062

Conclusions and Future Work

The objective of this paper is to study how to incorporate scripts that contain human knowledge into the tree policy of MCTS in order to improve its performance in the context of RTS games, while still maintaining the original search space of MCTS, and ensuring MCTS would converge to the optimal action in the limit. We presented two tree policies (FC-GNS and k-GNS) that do such integration, one using a single script and the other using a collection of scripts. We also reported our experiments on using the scripts in the playout policy.

The experimental results showed that both tree policies can improve the gameplay strength upon the baseline algorithm. They also scale well to larger maps (the larger the map, the larger the increase in performance with respect to NaïveMCTS), where the baseline algorithm struggles to perform. Additionally, experimental results on using scripts for the playout policy suggested that the bias should be carefully considered when designing scripts for playout policies.

This paper opens a wide range of possible future work. In particular we would like to study the effects of the ϵ parameter in k-GNS, which controls how much guidance is provided by the scripts. Additionally, another interesting problem is how to design a strong and less biased script for the playout policy. Finally, we want to compare with other ways to exploit scripts in the literature both from a practical and theoretical point of view.

References

Andersen, P.-A.; Goodwin, M.; and Granmo, O.-C. 2018. Deep rts: A game environment for deep reinforcement learning in real-time strategy games. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, 1–8. IEEE.

Barriga, N. A.; Stanescu, M.; and Buro, M. 2017. Combining strategic learning with tactical search in real-time strategy games. In *Proceedings of the Thirteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-17), October 5-9, 2017, Snowbird, Little Cottonwood Canyon, Utah, USA.*, 9–15.

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

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

Coulom, R. 2007. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, 72–83. Springer.

Justesen, N.; Tillman, B.; Togelius, J.; and Risi, S. 2014. Script-and cluster-based uct for starcraft. In *2014 IEEE Conference on Computational Intelligence and Games*, 1–8. IEEE.

Kocsis, L., and Szepesvári, C. 2006. Bandit based monte-carlo planning. In *European conference on machine learning*, 282–293. Springer.

Moraes, R. O., and Lelis, L. H. 2017. Asymmetric action abstractions for multi-unit control in adversarial real-time games. *arXiv preprint arXiv:1711.08101*.

Moraes, R. O.; Marino, J. R.; Lelis, L. H.; and Nascimento, M. A. 2018. Action abstractions for combinatorial multi-armed bandit tree search. In *Fourteenth Artificial Intelligence and Interactive Digital Entertainment Conference*.

Ontañón, S. 2016. Informed monte carlo tree search for real-time strategy games. In *Computational Intelligence and Games (CIG), 2016 IEEE Conference on*, 1–8. IEEE.

Ontañón, S. 2017. Combinatorial multi-armed bandits for real-time strategy games. *Journal of Artificial Intelligence Research* 58:665–702.

Ontañón, S.; Barriga, N. A.; Silva, C. R.; Moraes, R. O.; and Lelis, L. H. 2018. The first microrts artificial intelligence competition. *AI Magazine* 39(1).

Rosin, C. D. 2011. Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence* 61(3):203–230.

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

Silver, D., and Tesauro, G. 2009. Monte-carlo simulation balancing. In *Proceedings of the 26th Annual International Conference on Machine Learning*, 945–952. ACM.

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

Sun, P.; Sun, X.; Han, L.; Xiong, J.; Wang, Q.; Li, B.; Zheng, Y.; Liu, J.; Liu, Y.; Liu, H.; and Zhang, T. 2018. Tstarbots: Defeating the cheating level builtin AI in starcraft II in the full game. *CoRR* abs/1809.07193.

Synnaeve, G.; Nardelli, N.; Auvolat, A.; Chintala, S.; Lacroix, T.; Lin, Z.; Richoux, F.; and Usunier, N. 2016. Torchcraft: a library for machine learning research on real-time strategy games. *arXiv preprint arXiv:1611.00625*.

Tian, Y.; Gong, Q.; Shang, W.; Wu, Y.; and Zitnick, C. L. 2017. Elf: An extensive, lightweight and flexible research platform for real-time strategy games. *Advances in Neural Information Processing Systems (NIPS)*.

Vinyals, O.; Ewalds, T.; Bartunov, S.; Georgiev, P.; Vezhnevets, A. S.; Yeo, M.; Makhzani, A.; Küttler, H.; Agapiou, J.; Schrittwieser, J.; et al. 2017. Starcraft ii: a new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*.