Parametric Action Pre-Selection for MCTS in Real-Time Strategy Games *

Abdessamed Ouessai[1], Mohammed Salem[1], and Antonio M. Mora[2]

[1] Dept. Computer Sciences, University of Mascara, Mascara, Algeria
abdessamed.ouessai@univ-mascara.dz — salem@univ-mascara.dz
[2] Dept. Signal Theory, Telematics and Communications, ETSIIT-CITIC, University of Granada, Granada, Spain
amorag@ugr.es

Abstract. The core challenge facing search techniques when used to play Real-Time Strategy (RTS) games is the extensive combinatorial decision space. Several approaches were proposed to alleviate this dimensionality burden, using scripts or action probability distributions, based on expert knowledge. We propose to replace expert-authored scripts by a collection of smaller parametric scripts we call heuristics and use them to pre-select actions for Monte Carlo Tree Search (MCTS). The advantages of this proposal consist of granular control of the decision space and the ability to adapt the agent's strategy in-game, all by altering the heuristics and their parameters. Experimentation results in μRTS using a proposed implementation have shown a significant performance gain over state-of-the-art agents.

Keywords: Game AI · Real-Time Strategy · MCTS · μRTS

CEUR-WS.org/Vol-2719/paper11.pdf

1 Introduction

Video games belonging to the Real-Time Strategy (RTS) sub-genre can be viewed as an evolution of classic board games. Games such as Chess, Checkers, and Go depict an abstract conflict situation between two parties. The capabilities of modern computing devices, usually paired with advanced game-engines, allow RTS games to portray concrete conflict situations, approximately simulating the dynamics of real-world military disputes. Players in an RTS game can simultaneously control multiple entities (units) in real-time, within a large, dynamic, and uncertain environment.

Designing a successful artificial RTS player is a demanding task that requires overcoming many challenges. In particular, the large combinatorial decision and state spaces constitute a significant bottleneck for game-playing agents based on search or machine learning. Task decomposition is a common solution to mitigate

* This work has been supported in part by projects B-TIC-402-UGR18 (FEDER and Junta de Andalucía), RTI2018-102002-A-I00 (Ministerio Español de Ciencia, Innovación y Universidades), projects TIN2017-85727-C4-1-2-P (Ministerio Español de Economía y Competitividad), and TEC2015-68752 (also funded by FEDER).

Copyright © 2020 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

the domain's complexities, usually by defining multiple sub-tasks in each level of abstraction. Decision making in an RTS game flows through three levels of abstraction: strategic, tactical, and reactive-control [14].

Monte Carlo Tree Search (MCTS) is a sampling-based search approach with a successful track record in many board games, especially those known for their high branching factor such as Go. MCTS was also adapted for use as a holistic agent in RTS games, but it still struggles to replicate the same degree of success as in Go, due to the branching-factor bottleneck. The proposed improvements usually fall in two main scopes: (1) The use of expert-knowledge to abstract the decision space and lower the branching factor, and (2) the use of MCTS solely as a component dedicated to smaller tactical and reactive situations, as part of an agent. While there are efforts to combine both scopes [10], the reliance on a fixed set of expert-authored scripts remains a vector of exploitation.

In this paper, we propose an action pre-selection algorithm that relies on the building blocks of expert scripts rather than full scripts. Expert-authored scripts combine several smaller scripts, we call heuristics, to form a scripted agent. A heuristic represents an isolated task performed by a unit or a group of units, such as harvesting or attacking. Our algorithm depends on parametric heuristics to generate a wide variety of scripts that can be used to pre-select actions for predefined groups of units, to feed into an MCTS agent. The heuristics' parameters can be modified in real-time to adjust both the decision granularity and the adopted strategy, which opens the perspectives for dynamic strategy adaptation through MCTS. Experimentation results in the μRTS test-bed reveals a significant MCTS performance gain related to the reduced branching factor and the more focused decision space.

In the next section, we will review some preliminaries about RTS games and MCTS. In Section 3 we will explore some relevant works related to our proposal, and in Section 4 we will detail the theoretic and implementation aspects of our approach. Experimentation results will be presented and discussed in Section 5, followed by a conclusion in Section 6.

2 Background

2.1 RTS Games

An RTS game is a zero-sum, multiplayer, non-deterministic, and imperfect-information game, where players compete for resources to defeat an opposing side using military force. To win, the player is expected to devise and execute a strategy that can overcome his opponent and ultimately eliminate all his units. The player manages his units by issuing a unit-action to each, forming a player-action in each decision. The game's environment usually consists of a large map with topographic features, covered by a fog-of-war layer that reduces visibility.

RTS games gained significant popularity on the PC platform thanks in part to their PC-friendly user interfaces and control schemes, reminiscent of a typical PC software. The most commercially successful RTS titles include StarCraft, Command & Conquer, and Age of Empires. Enabling AI research

on commercial games started as an independent community effort that resulted in several unofficial APIs, such as BWAPI and Wargus. Much later, Blizzard provided an official StarCraft II AI API and toolset in collaboration with DeepMind[3]. Independent RTS AI research platforms have also emerged, such as ORTS, μRTS, ELF, and DeepRTS.

We used μRTS [12] as our experimentation test-bed. Conceived by Santiago Ontañón, μRTS is the most lightweight and accessible RTS AI research platform [15]. It features a minimalistic RTS implementation focused on the most fundamental RTS challenges and includes an efficient forward model necessary for implementing lookahead-based approaches.

Formally, an RTS game can be defined as a tuple $G = (S, A, P, \tau, L, W, s_{init})$ [12], where S, A, and P represent the state space, the player-action space, and the players' set, respectively. The transition function, $\tau : S \times A \times A \rightarrow S$, takes a game state at time t and the player-action of each player (assuming two-player setting) and returns a new game state at time $t+1$. Function $L : S \times A \times P \rightarrow \{true, false\}$ returns *true* when a player-action considered in a given game state by a given player is legal. Function $W : S \rightarrow P \cup \{ongoing, draw\}$ returns the winner of the game, if any, or whether the game is still ongoing or is a draw. s_{init} is the initial game state.

2.2 MCTS

MCTS [5] is a family of anytime (yields a solution under any amount of processing time) sampling-based search algorithms applicable to Markov decision processes (MDPs) [17]. They work by incrementally constructing a game tree across multiple iterations. A single MCTS iteration consists of four basic phases. The first phase, *selection*, employs a tree-policy to select which tree node to expand. Next, an *expansion* phase creates and appends a new node to the selected node. A *simulation* (or *playout*) is then executed starting from the new node using a default-policy and lastly a *backpropagation* phase uses the simulation's outcome to update the reward estimates and the visit count of all nodes on the way up to the root. The action leading to the most visited node is usually the one returned.

Upper Confidence Bounds for Trees (UCT) [7] is a popular MCTS algorithm that frames the selection phase as a Multi-Armed Bandit (MAB) [1] problem, then uses the UCB1 formula to select nodes for expansion. UCT works well in high branching-factor domains, such as Go, but suffers greatly when the decision space has also a combinatorial structure, as in RTS games. This drawback is due to UCB1's requirement to explore all possible moves at least once to commence exploitation. The short decision cycle and huge average number of possible moves at a decision point in RTS games do not allow UCT the chance to explore all moves.

NaïveMCTS [12] addressed UCT's shortcomings in combinatorial decision spaces by framing the selection phase as a Combinatorial MAB (CMAB) prob-

[3] https://github.com/deepmind/pysc2

lem. This change highlights the reward contribution of each component (unit-action) of a decision (player-action) and makes it possible to employ a naïve sampling strategy. Naïve sampling assumes that the reward estimates of unit-actions can be summed to obtain the reward estimate of the player-action they form. The problem is thus subdivided into n local MABs (n: the number of units) and one global MAB. NaïveMCTS does not need to explore all moves before exploitation since it relies on an ϵ-greedy strategy instead of UCB1.

3 State of the Art

Managing the complex decision-space of RTS games in the context of search is usually done by defining a proxy layer that guides search towards high-value actions while ignoring the rest. This layer is defined by expert knowledge and may take the form of expert-authored scripts, or action probability distributions learned from expert traces. State abstraction through unit clustering is also a way to further reduce the branching factor by considering clusters of units instead of individual units.

In RTS combat scenarios, Balla and Fern [2] defined a pair of abstract actions and a unit grouping approach to facilitate search using UCT. Justesen et al [6] used UCT in the space of actions proposed by scripts and further simplified search using unit clustering by K-means. For full RTS scenarios, Barriga et al [4] proposed to configure scripts with exposed choice points and use UCT to plan in the space of choice points. In [10], Moraes and Lelis combined NaïveMCTS with the concept of asymmetric abstraction to search in a decision space where each group of units draws its actions from a distinct set of scripts, or the low-level actions set. Scripts were also used to bias the selection phase of NaïveMCTS in [19]. Inspired by AlphaGo, Ontañón [13] trained a Bayesian model from expert traces to guide NaïveMCTS selection.

Notable non-MCTS abstraction-based search algorithms include Stratified Strategy Selection (SSS) [8], which uses a type system for unit clustering, and hill-climbing to search for an optimal script for each cluster. The main drawback of action-abstraction approaches is their reliance on a fixed set of scripts that may produce a predictable agent. In [18], the authors propose to generate a larger set of scripts from a small initial set through a voting scheme. Marino et al [9] sought to find an optimal action abstraction by evolving a large pool of scripts generated by varying the parameters of predefined rule-based scripts.

Planning exclusively in a scripts-induced decision space can compromise low-level tactical performance. For this reason, several works paired action abstraction with low-level search. StrategyTactics [3] uses a pre-trained model based on Puppet Search for strategic planning and NaïveMCTS for tactics. Similarly, Neufeld et al [11] used NaïveMCTS for tactical planning in their HTN (Hierarchical Task Networks)-MCTS hybrid agent. In [18], the authors used ABCD (AlphaBeta Considering Duration) for tactical decisions, in a limited game state.

We propose a parametric action pre-selection scheme that, in contrast with previous approaches, relies on heuristics instead of full scripts. Our approach

works by altering the decision space for NaïveMCTS, according to a set of provided parameters that govern the possible strategic and tactical decisions. Moreover, our approach supports assigning heuristics to individual units or groups of units. The advantage of this approach is its ability to implement a wide spectrum of strategies by adjusting the heuristics' parameters pre-game or in-game.

4 Parametric Action Pre-Selection

An expert-authored strategy or script in an RTS game can be broken down into a set of heuristics, each controlling a group of units. Changing or replacing one heuristic can affect the goal and the performance of the strategy with varying degrees. If we assume that all the possible RTS heuristics are known, then it becomes possible to generate all RTS strategies by combining heuristics. An agent can adapt its strategy according to its opponent or environment by replacing problematic heuristics. To preserve tactical performance, a heuristic can delegate all tactical choices to a search approach.

We suggest that one way to capture a broad spectrum of all possible heuristics is by defining parametric heuristics. Each parameter may govern an aspect of the heuristic, creating multiple possible heuristics with each different parameter value combination. Therefore, the combination of heuristics and their parameters define a strategy. A heuristic can be expert-authored or automatically learned.

Formally, we define a heuristic $h \in \mathcal{H}$, where $\mathcal{H}$ is the set of heuristics, as a function $h : S \times \mathcal{U} \times \mathcal{A}^l \times \mathcal{R}_h \rightarrow \mathcal{A}^k$ taking a game state $s \in S$, a unit $u \in \mathcal{U}$, all legal unit-actions $(a_1, \cdots, a_l) \in \mathcal{A}^l$ possible for u in s, and a parameter vector $\mathbf{p} \in \mathcal{R}_h$ as input to produce a unit-action tuple $\alpha = (a_1, \cdots, a_k) \in \mathcal{A}^k$ where $\mathcal{A}^k \subseteq \mathcal{A}^l$ and $k \leq l$. The sets $\mathcal{U}$, $\mathcal{R}_h$ and $\mathcal{A}$ represent the set of units, parameter vectors of heuristic h, and legal unit-actions, respectively.

In hard-coded scripts, parameter vectors are usually constant, but in our proposed approach, the parameter vector of a heuristic can be variable. A heuristic h is fully deterministic if $k = 1$ and no random aspect intervenes in its execution. In case $k > 1$, a search algorithm can select an optimal action from the heuristic's output, α. A heuristic may employ any suitable algorithm, including pathfinding, to narrow down the number of actions in α and reduce the overall branching factor.

For instance, a common heuristic in RTS games is the *harvest* heuristic which when applied to a *Worker* unit discards all unit-actions in favor of those that guide the *Worker* back and forth between a *resource deposit* (harvest) and a *Base* (return). *harvest* may expose parameters such as the maximum resources to harvest or the pathfinding algorithm to use.

A heuristic h can be associated with a group of units $g \in \mathcal{P}(\mathcal{U})$, in which case it will be applied to each unit in g under parameter vector $\mathbf{p}$, and we denote it $h[g, \mathbf{p}]$. We define $\mathcal{D}$ as the set of all possible unit partitionings, where a partitioning $d \in \mathcal{D}$ can be defined as $d = \{g_1, \cdots, g_m \mid g_i \in \mathcal{P}(\mathcal{U})\}$.

An action pre-selection process $\mathcal{T}(s, \mathcal{U}, \mathcal{A}_0, x_1, \cdots, x_n)$ is an n-phase algorithm where each phase $x_i(\mathcal{A}_{i-1}, d_i, \mathcal{H}_i, \theta_i)$ consists of a unit partitioning scheme

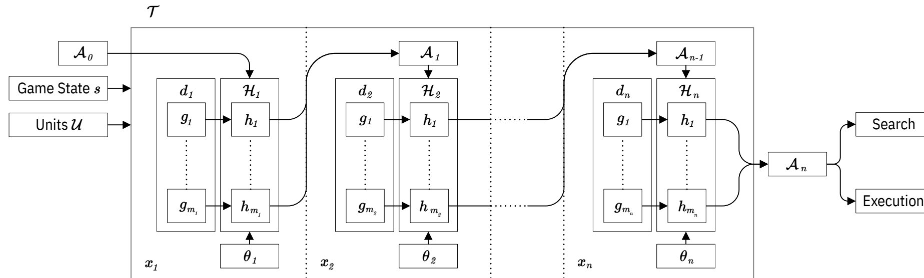

Fig. 1. The action pre-selection process.

$d_i \in D$ that generates a set of m_i unit groups g_j for which a heuristic $h_j[g_j, \mathbf{p}_j] \in \mathcal{H}_i(\mathcal{H}_i \subset \mathcal{H})$ is applied $\forall j \in 1, \cdots, m_i$, under the parameter vector $\mathbf{p}_j \in \theta_i$. Each pre-selection phase operates on the unit-actions output of the previous phase, $\mathcal{A}_{i-1}$, and the initial phase operates on $\mathcal{A}_0$, the set of all legal unit-actions of the units in $\mathcal{U}$. The output of phase x_i consists of the set of unit-actions $\mathcal{A}_i$ calculated by the heuristics of $\mathcal{H}_i$ for each unit in $\mathcal{U}$. The final output of $\mathcal{T}$ is the unit-action set $\mathcal{A}_n$. Figure 1 illustrates the pre-selection process.

Intuitively, an action pre-selection process is a successive refinement strategy that works by manipulating the set of unit-actions possible for each unit in a game state according to a global strategy expressed by the set of heuristics, partitionings, and parameters defined for each pre-selection phase. The resulting set of unit-actions represent a decision to execute, or the possible options admissible by the global strategy. In the latter case, a search algorithm such as MCTS can be employed to find an optimal player-action in accordance with the global strategy, in a much smaller and focused decision space. A global strategy is represented by $\sigma_n = (d_1, \cdots, d_n, \mathcal{H}_1, \cdots, \mathcal{H}_n, \theta_1, \cdots, \theta_n)$ in an action pre-selection process $\mathcal{T}$.

As an example, it is possible to define a hierarchical 2-phase pre-selection process, where in the first phase the units are split into two large groups $d_1 = \{defense, offense\}$ and are assigned to two heuristics $\mathcal{H}_1 = \{defend, attack\}$ under parameters θ_1. In the second phase, the units could be split into more specialized groups such as: $d_2 = \{baseDef, barracksDef, offense\}$. The first phase assures a common behavior for defense units, and the second phase builds on that to create specialized defense units. We will describe an action pre-selection implementation proposition for RTS games next.

4.1 Implementation

We propose an action pre-selection implementation for RTS games that delegates tactical decision making to NaïveMCTS and allows the expression of strategic decisions through parametric heuristics. Our implementation, *ParaMCTS*, relies on a 2-phase action pre-selection process that partitions units into four *functional*

groups in the first phase, and into two *situational* groups in the second phase. The intuition behind both group types comes from the way human RTS players attribute different *functions* to different unit groups while macro-managing, and how they switch to micro-management for units in conflict *situations*. We define d_1 and d_2, in the context of μRTS, as such:

d_1: Functional groups

- *Harvesters:* Worker units dedicated to gathering resources and building structures.
- *Offense:* Mobile units with the purpose of assaulting opponent units.
- *Defense:* Mobile units assigned to defend the Base's perimeter.
- *Structures:* Barracks and Base. Responsible for producing mobile units.

d_2: Situational groups

- *Front-Line:* Mobile units in close-contact with opponent units.
- *Back:* All units not in the Front-Line group.

d_1 assigns each unit to its relevant group based on a predefined unit-composition that declares the maximum count of each unit-type to be found in each group. For instance, it is possible to specify the maximum number of Workers in the *Harvesters*, *Offense* or *Defense* group. As for d_2, it populates the *Front-Line* group by selecting a number of units within the fire-range of opponent units, or those targeting an opponent unit. The unit-composition and front-line selection method are both partitioning parameters to provide. The heuristics in $\mathcal{H}_1$ and $\mathcal{H}_2$ are rule-based and are described as follows:

$\mathcal{H}_1$:

- **Harvest:** Applies to the *Harvesters* group. Automates the resource harvesting process and provides Barracks building options whenever possible. Parameters include the building location selection mode (isolated, random, ..., etc.) and the number of build options.
- **Attack:** Applies to the *Offense* group. Find and track opponent units for suppression. Parameters include the tracking mode (closest, minHP,...etc.), the maximum number of units to track, and the number of escape routes to consider in a close encounter.
- **Defend:** Applies to the *Defense* group. Remain within a defense perimeter around the Base and attack incoming opponent units. Parameters include the geometry and size of the defense perimeter, the defense mode, and the maximum number of units to attack.
- **Train:** Applies to the *Structures* group. Trains units following the unit-composition provided to d_1. Parameters include the unit-composition, the training mode (isolated side, random side, ..., etc), and the number of training options.

Note how in each heuristic a numeric parameter decides the number of options for certain actions (build, targets, ...). This type of parameter dictates how many choices (k) NaïveMCTS can operate on for each unit of the same group, which directly impacts the branching factor. **Harvest**, **Attack**, and **Defend** use a pathfinding algorithm to direct units towards their goals.

$\boldsymbol{\mathcal{H}_2}$:

- **Front-Line Tactics:** Applies to the *Front-Line* group. Reduces the Wait unit-action duration to increase the units' reactivity while in combat.
- **Back Tactics:** Applies to the *Back* group. Keeps the default Wait unit-action duration.

Note that *Front-Line* units do not depend on d_1 partitioning, meaning that any unit in a d_1 group can also be considered a *Front-Line* unit. In total 47 parameter was defined for all heuristics and partitionings. The full parameters list can be consulted in this approach's source code repository [4].

Switching heuristics on the fly is a way to adapt the agent's strategy according to changes in the overall situation. We propose to switch the heuristics of d_1's *Defense* group from **Defend** to **Attack** according to a conditional trigger. The switch is triggered whenever the score of the army composition is greater than the opponent's by a predefined margin we call the overpower factor. The score simply counts all mobile units and attributes greater weight to assault units.

Lastly, *ParaMCTS* uses a NaïveMCTS enhancement, previously conceived by the authors (to appear in [16]). This enhancement hard-prunes a portion of player-actions that include a Wait unit-action for the sake of decreasing the branching factor.

5 Experiments & Results

Using *ParaMCTS*, we conducted a series of experiments to gauge the benefits of action pre-selection on the performance of NaïveMCTS. Since the main effect of action pre-selection is a significantly reduced branching factor, we would like to test which MCTS parameter can exploit the downsized decision space to add the most value to performance. We focus on the two prominent search parameters, maximum depth, and playout duration. We experiment using the following sets of possible values for both parameters: $depthVals = \{10, 15, 20, 30, 50\}$ and $durationVals = \{100, 150, 200, 300, 500\}$.

We define $ParaMCTS(depth, duration)$ as the *ParaMCTS* variant using both *depth* and *duration* as the maximum search depth, and playout duration, respectively. All experiments were run on two PCs with relatively similar processors using the latest version of μRTS as of 10 July 2020. Each agent was attributed a $100ms$ computation budget, similarly to the μRTS competition setting, and each experiment was repeated on three maps representing increasingly larger

[4] https://github.com/Acemad/UMSBot

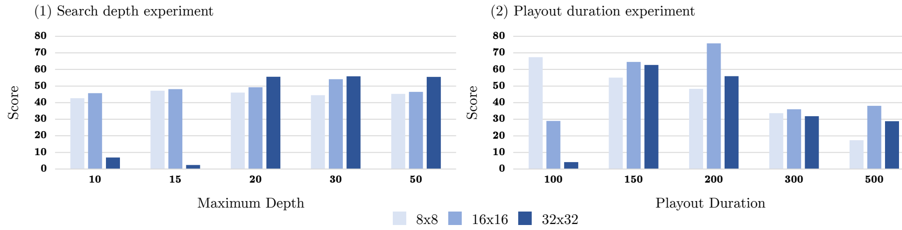

Fig. 2. The results obtained by each *ParaMCTS* variant in both tournaments of Experiment 1. The score represents the win rate of each variant against the other variants.

Table 1. Experiment 2 results. Each square represents the score obtained by *ParaMCTS*(*depth*, *duration*) against *MixedBot*. Cells with a score above 50 are gradually saturated in four levels: 50-59, 60-69, 70-79, and 80-89

	Playout Duration																	
	8 x 8						16 x 16						32 x 32					
Max Depth	10	150	200	300	500	Avg	100	150	200	300	500	Avg	100	150	200	300	500	Avg
10	74.5	69.5	59	46	19	53.6	3	74	63	20	20	36	68.5	54.5	35.5	33	20.5	42.4
15	82	65	56	46	28.5	55.5	1	80	71	10	28	38	72.5	44	42	34	25.5	43.6
20	86	74	51	40	23	54.8	88	77	59	15	25	52.8	65.5	50.5	46	39.5	17.5	43.8
30	81	68	62	40	18	53.8	71	73	66	9	21	48	74	46	43	38.5	25	45.3
50	83.5	68	69	46	21	57.5	68	74	62	4	29	47.4	76.5	41	46.5	45.5	24	46.7
Avg	81.4	68.9	59.4	43.6	21.9		46.2	75.6	64.2	11.6	24.6		71.4	47.2	42.6	38.1	22.5	

branching factors, namely *basesWorkers*8×8, 16×16, and 32×32. To guarantee a fair comparison, *ParaMCTS* was parameterized to adopt strategies similar to its opponents', even when there is a possibility for exploitation. In all experiments, we calculate the score of an agent likewise: $score = wins + draws/2$, and normalize it between 0 and 100.

5.1 Experiments 1 & 2: Search Depth and Playout Duration

To study the effect of various search depths and playout durations on *ParaMCTS* we ran 120 iteration of a round-robin tournament between each *ParaMCTS*(10, *duration*) variant for each *duration* in $durationVals$, and another 120 round-robin iteration between each *ParaMCTS*(*depth*, 100) variant for each *depth* in *depthVals*. Fixed values 10 and 100 for *depth* and *duration*, respectively, are the default NaïveMCTS values. Results of this experiment are shown in Figure 2.

In the second experiment, we took all the possible *depth* and *duration* combinations from $depthVals \times durationVals$ and ran 100 matches (switching sides after 50 matches) between each resulting *ParaMCTS*(*depth*, *duration*) and *MixedBot*, a state-of-the-art agent combining various techniques [9] [8] [10] [3]. The results of this experiment in the three maps are presented in Table 1.

From the results of both experiments, we can see how overall *ParaMCTS* performance seems to be particularly sensitive to the playout duration. In Fig-

Table 2. Overall results of Experiment 3 tournament, in all maps. Row vs Column.

	ParaMCTS	MixedBot	Izanagi	Droplet	NMCTS*	NMCTS	Average
ParaMCTS	-	84.8	50.0	72.0	96.0	94.7	79.5
MixedBot	22.8	-	32.8	53.5	96.8	96.0	60.4
Izanagi	50.7	64.8	-	42.7	89.5	90.2	67.6
Droplet	30.2	45.0	53.3	-	89.7	90.2	61.7
NMCTS*	7.3	2.2	9.2	7.2	-	50.2	15.2
NMCTS	6.8	2.5	9.7	8.7	47.7	-	15.1

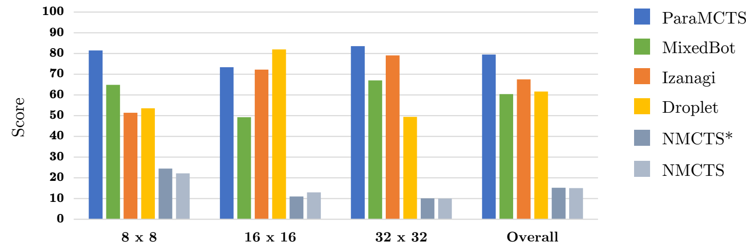

Fig. 3. The scores obtained by each agent in each map in Experiment 3 tournament.

ure 2-(2) performance vary significantly between the playout durations. In the smallest maps, short playouts work best, but in larger maps, slightly longer playouts work well up to a certain threshold. As for search depth, it is clear that in the largest map a deeper search yields the most benefit, as seen in Figure 2-(1). As for the small and medium maps, deeper search holds fewer benefits.

Against *MixedBot* (Table 1), the best results are obtained when the playout duration equals 100 cycles, in all three map sizes, even if 150 cycles seem promising for the 16×16 map. When looking at the search depth, going down 20 levels in the tree is the optimal depth for 8×8 and 16×16 maps. In the 32×32 map, searching as deep as 50 levels yield the best performance. Clearly, deeper search yields the most performance increase than longer playouts. We believe this is true because deeper search may be responsible for more accurate player-action reward estimates, due to the increased number of visited nodes, and playouts, towards the depth of the game tree. On the other hand, longer playouts yield a lower number of visited nodes, and playouts, which could negatively impact performance. Larger maps benefit the most from deeper search because the map's dimensions contribute to the sparsity of rewards, and a deeper search can reach rewarding states more frequently.

5.2 Experiment 3: Comparison Against State-of-the-Art

To compare the overall performance of *ParaMCTS* against the current best performing agents, we took three top ranking agents from 2019's μRTS competition: *MixedBot*, *Izanagi*, [10] [9] and *Droplet* [19], and performed a 100-iteration

round-robin tournament between them, *ParaMCTS* (with the optimal depth and duration values found in previous experiments, in each map), NaïveMCTS (as a baseline) and a NaïveMCTS variant, NMCTS*, using the same search depth and playout duration as *ParaMCTS*. Results of the tournament are presented in Table 2 and Figure 3.

In terms of overall performance, *ParaMCTS* outperformed all state-of-the-art agents by a comfortable margin. *ParaMCTS* was able to achieve an 11.9 points margin over the 2nd best agent, *Izanagi*, and 19.1 points margin over the 4th best, *MixedBot*. Both agents make use of a combination of advanced techniques. This result is direct evidence of the potency of our action pre-selection approach when coupled with NaïveMCTS. In individual maps, *ParaMCTS* outperformed the other agents in 8×8 and 32×32 maps, but it was outdone in the 16×16 map by *Droplet*. This can be explained by the adoption of an opportunistic strategy by *Droplet* in the 16×16 map. Although *ParaMCTS* can be easily configured to adopt such strategies, we chose not to do so to keep the comparison as fair as possible. NMCTS* did not offer any tangible performance gain over NaïveMCTS, which indicates that increasing the search's depth will not yield any performance gain if not paired with a significant decision-space reduction.

6 Conclusions & Future Work

We have presented an integrated action and state abstraction process that partitions units into multiple groups, and assigns heuristics to each group for the sake of obtaining a downsized set of pre-selected unit-actions. The proposed process receives a collection of parameters that define the partitioning and heuristics. It is possible to alter the heuristics parameters in-game to adapt the agent's strategy in a granular fashion. We have proposed a theoretical definition and demonstrated how it can be implemented through a full RTS agent in μRTS. Experimentation results show a significant improvement over state-of-the-art μRTS agents. *ParaMCTS* will take part in the 4th μRTS competition (to be organized as part of IEEE CoG 2020), under the alias *UMSBot*.

The proposed action pre-selection implementation, *ParaMCTS*, is a single possibility among many. Using action pre-selection as a basis to develop more sophisticated agents is conceivable. Although proposed as an approach to lower the RTS decision space dimensionality, we believe this technique could be easily adapted to any multi-unit real-time game. Moreover, we believe that this technique can also provide difficulty adjustment through parameter optimization.

As for future works, we are interested in auto-adapting the heuristics and their parameters on the fly, using opponent modeling in different environments. We are also looking into applying evolutionary algorithms to evolve and find optimal strategies. Automatically learning new heuristics and unit partitionings in the context of action pre-selection is also an interesting direction.

References

1. Auer, P., Cesa-Bianchi, N., Fischer, P.: Finite-time Analysis of the Multiarmed Bandit Problem. Machine Learning **47**(2/3), 235–256 (2002)
2. Balla, R.K., Fern, A.: UCT for Tactical Assault Planning in Real-Time Strategy Games. In: Proceedings of IJCAI'09. pp. 40–45 (2009)
3. Barriga, N.A., Stanescu, M., Besoain, F., Buro, M.: Improving RTS Game AI by Supervised Policy Learning, Tactical Search, and Deep Reinforcement Learning. IEEE Comput. Intell. Mag. **14**(3), 8–18 (2019)
4. Barriga, N.A., Stanescu, M., Buro, M.: Puppet Search: Enhancing Scripted Behavior by Look-Ahead Search with Applications to Real-Time Strategy Games. In: AIIDE'15. pp. 9–15. Santa Cruz, California (2015)
5. Browne, C.B., Powley, E., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S.: A Survey of Monte Carlo Tree Search Methods. IEEE Trans. Comput. Intell. AI in Games **4**(1), 1–49 (2012)
6. Justesen, N., Tillman, B., Togelius, J., Risi, S.: Script- and cluster-based UCT for StarCraft. In: 2014 IEEE CIG. Dortmund, Germany (2014)
7. Kocsis, L., Szepesvári, C.: Bandit Based Monte-Carlo Planning. In: Machine Learning: ECML 2006. pp. 282–293. Lecture Notes in Computer Science, Springer Berlin Heidelberg (2006)
8. Lelis, L.H.S.: Stratified Strategy Selection for Unit Control in Real-Time Strategy Games. In: Proceedings of IJCAI'17. pp. 3735–3741. Melbourne, Australia (2017)
9. Mariño, J.R.H., Moraes, R.O., Toledo, C., Lelis, L.H.S.: Evolving Action Abstractions for Real-Time Planning in Extensive-Form Games. In: Proceedings of the Conference on Artificial Intelligence (AAAI) (2018)
10. Moraes, R.O., Mariño, J.R.H., Lelis, L.H.S., Nascimento, M.A.: Action Abstractions for Combinatorial Multi-Armed Bandit Tree Search. In: Proceedings of the 14th AIIDE. AAAI Publications (2018)
11. Neufeld, X., Mostaghim, S., Perez-Liebana, D.: A Hybrid Planning and Execution Approach Through HTN and MCTS. In: The 3rd Workshop on Integrated Planning, Acting, and Execution - ICAPS'19. pp. 37–45 (2019)
12. Ontañón, S.: The Combinatorial Multi-Armed Bandit Problem and Its Application to Real-Time Strategy Games. In: Proceedings of the 9th AIIDE. pp. 58–64. AAAI Publications (2013)
13. Ontañón, S.: Informed Monte Carlo Tree Search for Real-Time Strategy games. In: 2016 IEEE CIG. Santorini, Greece (2016)
14. Ontañón, S., Synnaeve, G., Uriarte, A., Richoux, F., Churchill, D., Preuss, M.: A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft. IEEE Trans. Comput. Intell. AI in Games **5**(4), 293–311 (2013)
15. Ouessai, A., Salem, M., Mora, A.M.: Online Adversarial Planning in μRTS : A Survey. In: 2019 International Conference on Theoretical and Applicative Aspects of Computer Science (ICTAACS). Skikda, Algeria (2019)
16. Ouessai, A., Salem, M., Mora, A.M.: Improving the Performance of MCTS-Based μRTS Agents Through Move Pruning. In: IEEE CoG. Osaka, Japan (2020)
17. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. Prentice Hall, Upper Saddle River, New Jersey (2010)
18. Silva, C.R., Moraes, R.O., Lelis, L.H.S., Gal, K.: Strategy Generation for Multi-Unit Real-Time Games via Voting. IEEE Transactions on Games (2018)
19. Yang, Z., Ontañón, S.: Guiding Monte Carlo Tree Search by Scripts in Real-Time Strategy Games. In: Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment. vol. 15, pp. 100–106 (2019)