SAND2024-06138T

Leveraging Monte Carlo Tree Search to Improve Teaming Performance in Multi-Agent Adversarial Environments

A Thesis Presented to
The Academic Faculty

By

Matthew Connelly

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in Robotics
College of Engineering

Georgia Institute of Technology

May, 2024

Leveraging Monte Carlo Tree Search to Improve Teaming Performance in Multi-Agent Adversarial Environments

Approved by:

Dr. Anirban Mazumdar, Advisor
School of Mechanical Engineering
Georgia Institute of Technology

Dr. Jonathon Rogers
School of Aerospace Engineering
Georgia Institute of Technology

Dr. Kyle Williams
Pathfinder Technologies
Sandia National Laboratories

Date Approved: May 15, 2024

To my family, friends, and colleagues, who push me to be a better version of myself every day.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Anirban Mazumdar, for providing me with a wealth of opportunities throughout my time as a Master's student. Your mentorship, both through this research and in life, will stick with me forever.

Thank you to all past and present members of the Georgia Tech Dynamic Adaptive Robotic Technologies Lab and the Low Cost Aerial Autonomy VIP course. I want to especially thank Zachary Goddard, Alexander Gross, Kevin Choi, and Rithesh Rajasekar for their collaboration on this project. Your motivation and encouragement kept me going, and it has been a privilege to work with all of you.

I want to recognize Sandia National Laboratories for sponsoring this work and thank those who gave their time to provide me with technical expertise and feedback. Specifically, I would like to thank Dr. Julie Parish, Dr. Kyle Williams, Rachel Schlossman, and Dr. Caleb Peck for their involvement.

Thank you to my alma mater, Georgia Tech, for 6 years of tremendous growth and development. Your rigorous classes, supportive faculty, innovative students, and inclusive clubs have shaped my personal and professional journey. To me, you are home.

Most importantly, thank you to my family and friends. Without you, none of this is possible. You are the inspiration behind everything I do, and I am truly fortunate to have each and every one of you by my side.

Sincerely,

Matthew Connelly

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This thesis describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the thesis do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

This thesis extends Monte Carlo Tree Search which is predominantly used for 1 v 1 games to a multi-agent adversarial environment. To do so, we introduce two frameworks: Grouped Action Multi-Agent Monte Carlo Tree Search and Split Action Multi-Agent Monte Carlo Tree Search. The former considers each layer in the tree as a team of players while the latter considers each layer in the tree as an individual player. Both frameworks incorporate intelligent teaming, allowing a team of agents to overcome environment complexity. Though, there are pros and cons to each framework. Due to a low branching factor which results in earlier exploitation, our Split Action Multi-Agent Monte Carlo Tree Search is determined to be the superior framework for use in an adversarial environment.

Initially, this framework struggles with large amounts of players or low timeouts. If it does not achieve a minimum tree height, default actions are returned for a subset of the agents. Therefore, this thesis introduces Root Parallelization with Varying Agent Ordering, overcoming this framework's scalability issues. This method utilizes multiple root trees with differing agent orderings to increase the number of simulations run within a set timeout and provide each agent with an action for execution. This leads to large increases in win percentage for all scenarios.

All of the above experiments utilize a default action set. For our final set of experiments, we insert a proportional controller that exhibits offensive strategy. This proportional controller enables an agent to effectively track an adversary. This combo action set leads to increases in win percentage for all scenarios. To improve the results further, we delete some overlapping actions from the combo action set to create our compact action set. The compact action set has a lower branching factor, allowing us to get deeper into the tree. This increases the winning percentages even further.

Ultimately, we are able to increase the winning percentages from 25% to near 100% by utilizing our frameworks and associated methods.

CHAPTER 1

INTRODUCTION

Artificial intelligence (AI) is commonly used as the backbone for real-time decision making in dynamic environments. Here, the surroundings are constantly changing, making it essential to detect and react to incoming observations quickly. Board games, autonomous driving, and warehouse logistics are just a few examples of where AI decision making is used today. In these environments, mistakes vary in severity with the most severe outcome occurring in autonomous driving. A mistake in this setting can lead to a loss of life. Similarly, AI decision making can be used in adversarial environments, such as aerial combat. In adversarial environments, adversaries oppose the agents in their pursuit of a specific objective. This poses a difficult problem where agents need to consider all sorts of external factors in the decision making process.

There are several techniques used to solve these adversarial environments. The most common technique is deep reinforcement learning (Deep RL). Deep RL makes use of deep neural networks where the input is the important environment states. Based on these states, the deep neural network will predict actions for the agents that maximize the discounted sum of future rewards. Adversarial environments are innately continuous. To handle an environment with continuous state and action spaces, policy-based Deep RL algorithms have been used [1, 2, 3].

Recently, these algorithms have been applied to complex aerial combat situations [4, 5, 6]. They have also been extended to the multi-agent setting. For example, multi-agent reinforcement learning (MARL) helped AlphaStar reach the level of Grandmaster in StarCraft II [7]. Similar to the environment depicted in this thesis, MARL has been used to tackle multi-agent aerial combat challenges [8]. One limitation is that these algorithms require an extensive training process, and if any environment conditions change, such as varying the parameters of the players, we need to re-train the model from scratch. This limits its usability in situations where we do not know the number, capabilities, or strategies of our adversaries.

A game tree search algorithm known as Monte Carlo Tree Search (MCTS) [9, 10] does not

have this limitation. MCTS does not have a training step. Instead, it investigates a set of actions from different game states, evaluating their outcomes through random simulations. To aid in the efficiency of this search, MCTS spends more time simulating and evaluating actions that led to good performance in prior simulations. This asymmetric tree-building [11] is due to MCTS' selection criteria known as the upper confidence bounds applied to trees (UCT) which exhibits a phenomenon known as the exploration vs. exploitation trade-off. Another benefit of MCTS is its anytime [12] nature. MCTS can be stopped at anytime, and a decision will still be provided for execution. This allows MCTS to be used in real-time settings, making it better suited than other game tree search algorithms like minimax [13].

Parallelization is a common technique used to improve the performance of vanilla MCTS. The effectiveness of MCTS is often limited by the number of simulations that can be performed within a specified timeout. Multi-threading and multi-processing have been shown to boost the number of simulations, leading to improvements in overall performance. The three primary types of parallelization that have been explored are leaf parallelization, root parallelization, and tree parallelization. This thesis focuses on root parallelization due to it achieving the highest performance for the program MANGO [14]. Root parallelization has multiple root trees running simultaneously, and their results are combined via an aggregation function. Recently, these parallelization techniques have been extended to the continuous domain. For example, both root and leaf parallelization have been used for decentralized multi-agent cooperative environments, such as trajectory planning for autonomous vehicles [15, 16]. As will be shown at the end of this section, our work builds on these papers, introducing a novel root parallelization technique for solving complex multi-agent adversarial environments.

The popularity of MCTS can be traced to its use in the game of Go. Google DeepMind combined MCTS with deep neural networks to defeat a professional Go player for the first time [17, 18, 19]. MCTS has also been applied to a wide variety of other games, such as poker [20], Pac-Man [21], StarCraft [22], and Hearthstone [23]. As stated previously, adversarial environments are innately continuous whereas board games like Go have discrete state and action spaces. More

recently, MCTS was applied to a single-agent adversarial environment with continuous state and action spaces [24] using the Maneuver Automaton (MA) [25, 26]. Our approach implements this prior work's planar 2D adversarial environment for testing in the multi-agent domain, incorporating exact environment dynamics and a similar reward function.

Another notable difference is our use of forward-simulated actions instead of motion primitives. Our approach utilizes a similar set of actions to [27, 28] which can be represented as maintaining constant control inputs of rudder deflection and throttle for a set amount of time. In terms of novel contributions, this thesis proposes three key advancements. The first contribution is two multi-agent MCTS frameworks known as Grouped Action Multi-Agent Monte Carlo Tree Search (GAMA-MCTS) and Split Action Multi-Agent Monte Carlo Tree Search (SAMA-MCTS) which incorporate inherent strategy and coordination for planning in multi-agent adversarial environments with continuous state and action spaces. The second contribution is a root parallelization method known as Root Parallelization with Varying Agent Ordering (RP-VAO) which overcomes scalability issues found in our SAMA-MCTS framework that arise when increasing the number of players. As the number of players increases in SAMA-MCTS, more simulations are required to achieve consistent results. This is not possible in real-time settings. By varying the ordering of the agents over multiple trees, these challenges are mitigated. The third contribution is the insertion of closed-loop or feedback actions into our default action set. In doing so, we are introducing a level of adaptability, responsiveness, and strategy into the decision making process.

We demonstrate the improvements caused by our frameworks and associated methods by running numerous tests where a team of agents utilizing MCTS engage a team of higher-performing baseline adversaries.

CHAPTER 2

MULTI-AGENT MONTE CARLO TREE SEARCH FRAMEWORKS

Our frameworks extend MCTS to solve a multi-agent adversarial environment with continuous state (S) and action spaces. The continuous action space is discretized using planar 2D forward-simulated actions from [27, 28] to create what we refer to as our default action set. This default action set is compatible with our frameworks. The actions are made up of 2 control inputs, rudder deflection (RD) and throttle (T), which are held constant for a set action duration (t). There are 3 possible values for RD and 3 possible values for T. Therefore, there are exactly 9 actions defined by $RD \times T$ within the new discrete action space (DA). The possible values of RD are turn left, go straight, and turn right, and the possible values of T are no throttle, medium throttle, and high throttle. By concatenating actions together from this simple DA, we are able to perform strategic and complex maneuvers.

Within our 2D planar multi-agent adversarial environment, $\vec{x}_i$ denotes the current state of the ith player where $\vec{x}_i \in S$. In our set-up, $\vec{A} = (\vec{x}_1, \vec{x}_2, ..., \vec{x}_m)$ represents the states of all m agents, and $\vec{B} = (\vec{x}_{m+1}, \vec{x}_{m+2}, ..., \vec{x}_{m+n})$ represents the states of all n adversaries. Furthermore, π_i denotes the current action being executed by the ith player where $\pi_i \in DA$. Collectively, $\Pi = (\pi_1, \pi_2, ..., \pi_{m+n})$ represents the set of current actions being executed by all players. Each node of the tree is defined as $node = (\vec{A}, \vec{B}, \Pi, f, d, p, averageReward, visitCount)$, where f is the framework type, d is the current game depth, and p is either the team or player that $node$ will expand all possible actions for at d. The possible values of p are dependent on f and will be described further in the following two sections.

In adversarial environments, the engagement time can be large, meaning that the game will not terminate in the near future. MCTS frequently needs to return an action once a certain timeout is reached. For real-time decision making, this timeout is very small. If MCTS has to search to the end of these games to discover an outcome, an outcome might never be found. This would lead to an executed action that is random. Therefore, our frameworks make use of a look-ahead time (l)

which represents how many seconds into the future to simulate before trying a new combination of actions. To evaluate each combination of actions, we introduce an intermediary reward function that is computed after all players take a simulated action. Therefore, with $t = 1$ s and $l = 4$ s, the intermediary rewards will be computed $\frac{l}{t} = 4$ times. We aggregate the intermediary rewards together once l is reached and use this value to update the average reward values for all nodes along the path from the selected node back up to the tree's root node. We then use UCT [9] to select a new starting node.

The intermediary reward function is shown in Equation 2.1. It is split into a discrete damage reward weighted by w and a position shaping reward [24] weighted by $1 - w$. The value D in the discrete damage reward is either a 0 or 1, representing if a player on the opposing team is in the current player's cone of fire as shown in Figure 2.1. We include this reward type to emphasize players taking actions to damage and eventually eliminate players on the opposing team. On the other hand, the position shaping reward provides an overall score of how advantageous the agents' positions are compared to the adversaries as shown in Figure 2.2. θ represents the angle from the current player's nose to the opposing player in question. This reward is scaled by the euclidean distance, e, multiplied by a constant c. The most advantageous position for a player is one where it is directly behind and facing its opponent while the opponent faces directly away. For our games, we scale the discrete damage reward and position shaping reward to be $-m \leq R_{\text{int}} \leq m$ before combining them. Therefore, the user only needs to tune w. Optionally, these actions of duration t can be broken down further into multiple steps of length Δt, leading to greater precision but with more computational cost. These intermediary rewards can then be combined via the trapezoidal rule as shown in Equation 2.2. Finally, we take the sum to get a singular reward value for the trajectory through Equation 2.3. The values of t (action duration), l (look-ahead time), w (reward type weighting), and c (position shaping reward constant) along with the size of the cone of fire are up to the user's discretion. For our results, we used $t = 1$ s, $l = 4$ s, $w = 0.5$, $c = 0.1$, and a cone

of fire that is 5 m long with an angle value of $30°$.

$$R_{\text{int}}(\vec{A}, \vec{B}, d) = w * (\sum_{a=1}^{m} D_a - \sum_{b=m+1}^{m+n} D_b) + (1 - w) * \sum_{a=1}^{m} \sum_{b=m+1}^{m+n} \frac{0.5 * (\cos\theta_a - \cos\theta_b)}{1 + ce} \quad (2.1)$$

$$R_{\text{trap}}(\vec{A}, \vec{B}, d) = \text{trapz}(R_{\text{int}}(\vec{A}, \vec{B}), \Delta t) \quad (2.2)$$

$$R_{\text{total}}(\vec{A}, \vec{B}) = \sum_{d=1}^{l} R_{\text{trap}}(\vec{A}, \vec{B}, d) \quad (2.3)$$

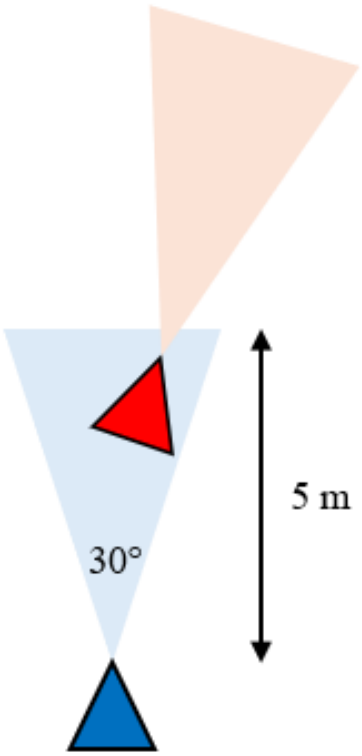

Figure 2.1: Adversary in agent's cone of fire

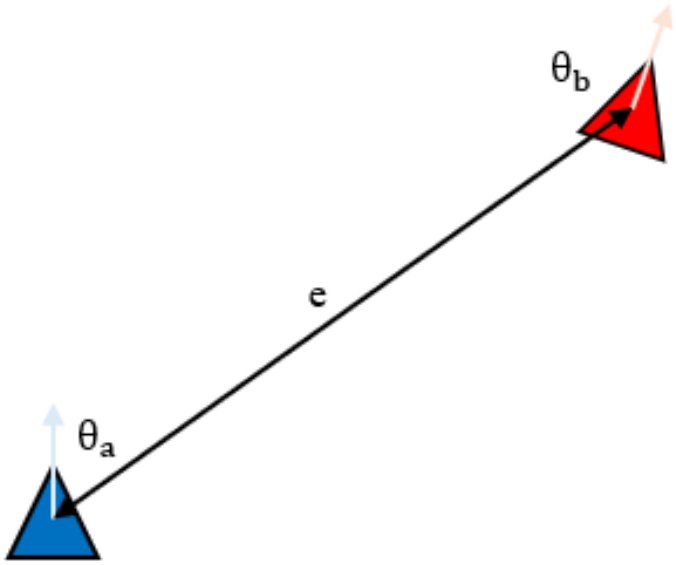

Figure 2.2: Important parameters for position shaping reward

The above reward calculation determines rewards for agents and will help us exploit actions for them that are deemed favorable by prior simulations. In MCTS, we also need to choose actions for adversaries. To build an effective tree, we need to exploit their best actions as well. This creates a realistic setting where the agents are seeing the most challenging version of the adversaries. To do this, we negate the stored rewards when it is an adversary's turn to choose an action. Therefore, the actions that are weighted more heavily by UCT are those that led to the worst agent reward which in turn are the best actions for the adversaries.

One complexity of multi-agent adversarial environments is that players on either team can be eliminated. To best utilize computational resources, it is crucial to only consider active players when running MCTS. Therefore, after executing an action, we update $\vec{A}$ and $\vec{B}$ to only include the states of the active agents and adversaries, respectively. Then, MCTS can be used to determine new actions for execution, and this process can be repeated.

Our frameworks make use of the following helper procedures shown in Algorithm 1 and described below which include slight variations to those found in Vanilla MCTS:

- *Selection*: This procedure starts at the tree's root node and utilizes UCT to traverse the tree. In our frameworks, traversing the tree involves iterating through all possible values for p at $d = 1$, incrementing d, and repeating the process until a leaf node is reached. Therefore, each subsequent value of p can be considered its own layer within the tree. Upon reaching a leaf node, this procedure calls *Expansion*.

- *Expansion*: This procedure expands the tree to include all possible actions for the leaf node's child or $p + 1$. As stated above, a new layer in the tree has been created. If $p + 1$ does not exist, d is incremented and p is set back to 1, meaning a new depth has been reached. This procedure calls *Simulation* from each of the newly added nodes which represents the set of all possible actions. This set of all possible actions is dependent on f.

- *Simulation*: This procedure forward-simulates out the rest of the engagement until reaching l. To do so, it uses the information stored in the node which called *Simulation*. This node has an associated depth value, d, and a set of actions being currently executed by all the players, Π. Before simulating out a set of actions, we must verify that Π does not contain any empty values. If π_p is empty for any value p, we provide a random action from p's action set. Additionally, for all future values of d until reaching l, we provide random actions for all p. Therefore, our rollout policy is entirely random. After each set of simulations, the reward is calculated using Equations 2.1 and 2.2. Upon reaching l, this reward is aggregated using Equation 2.3 and is passed to *Backpropagation*.

- *Backpropagation*: This procedure uses the reward found in *Simulation* to update the average reward values for all nodes along the path from the newly added node created in *Expansion* up to the tree's root node. At the conclusion of this procedure, we check to see if the timeout to return an action has been reached. If it has not, *Selection* resumes.

Algorithm 1 Multi-Agent Monte Carlo Tree Search Frameworks

procedure MCTS($\vec{A}, \vec{B}, f, l, timeout$)
    Input: Initial states $(\vec{A}, \vec{B})$, framework type f, look-ahead time l, $timeout$
    $root \leftarrow (\vec{A}, \vec{B}, \Pi = Empty, f, d = 0, p = 1, averageReward = 0, visitCount = 0)$
    while $timeout$ not exceeded **do**
        Selection($root, l$)
    end while
    return $\Pi^* = (\pi_1^*, \pi_2^*, ..., \pi_m^*)$ ▷ Dependent on f
end procedure
procedure Selection($node, l$)
    while $node$ is not a leaf node **do**
        $node \leftarrow$ UCT($node$)
    end while
    Expansion($node, l$)
end procedure
procedure Expansion($node, l$)
    if $node.d + 1 \leq l$ **then**
        for all possible actions of $node$'s child **do** ▷ Dependent on f
            $child \leftarrow$ InitializeChild($node$)
            Simulation($child, l$)
        end for
    end if
end procedure
procedure Simulation($node, l$)
    $start \leftarrow node$
    $reward \leftarrow 0$
    while $node.d \leq l$ **do**
        if any $node.\Pi$ is Empty **then**
            $node.\Pi \leftarrow$ RandomActions($node$)
        end if
        $reward \leftarrow reward +$ ForwardSimulate($node$)
        $node.d \leftarrow node.d + 1$
    end while
    return Backpropagation($start, reward$)
end procedure
procedure Backpropagation($node, reward$)
    while $node$ is not $root$ **do**
        $node.averageReward \leftarrow$ UpdateAverage($node.averageReward, reward$)
        $node.visitCount \leftarrow node.visitCount + 1$
        $node \leftarrow$ GetParent($node$)
    end while
end procedure

A simple example depicting these steps for a 1 v 1 adversarial environment is shown in Figure 2.3. The true layer width is omitted to save space.

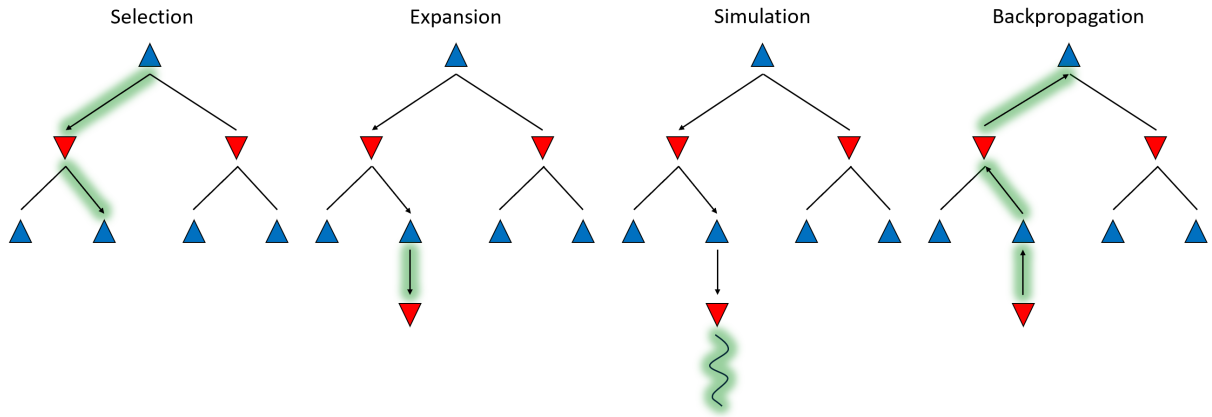

Figure 2.3: MCTS steps for a 1 v 1 adversarial environment with simplified layer width

2.1 GROUPED ACTION MULTI-AGENT MONTE CARLO TREE SEARCH

In the above section, it was stated that the possible values of p and their associated actions were dependent on f. Our first framework is known as Grouped Action Multi-Agent Monte Carlo Tree Search (GAMA-MCTS). This framework considers each node in a specific layer of the tree as a team of players, either agents or adversaries. GAMA-MCTS is comparable to vanilla MCTS.

The possible values of p are 1 and 2. To search t into the tree, we would need to look 2 layers deep. If $p = 1$, the possible actions come from the permutations of each agent's DA or $DA_1 \times DA_2 \times ... \times DA_m$. On the other hand, if $p = 2$, the possible actions come from the permutations of each adversary's DA or $DA_{m+1} \times DA_{m+2} \times ... \times DA_{m+n}$. As a result, the branching factor for the agents is 9^m and the branching factor for the adversaries is 9^n. In other words, the branching factor of the tree is exponential with respect to the number of agents or adversaries. This framework is limited in that it requires the duration of all π_i be the same value t. In addition, it requires that each combination of agents' or adversaries' actions be tried before exploiting for promising actions since they are considered as one. Though, when it comes time to determine actions for execution, the quality of all selected agents' actions will be equivalent.

The process to determine actions for execution is shown in Equation 2.4 where $root$ is the tree's

root node. Since all the agents' actions are combined, we only need to look at the first layer of the tree to determine the predicted optimal actions. A simple example of this framework's tree being built for a multi-agent adversarial environment with 2 agents and 1 adversary is shown in Figure 2.4. The true layer width is omitted to save space.

$$\Pi^* = \underset{child \in \text{GetChild}(root)}{\text{argmax}} \quad child.averageReward \tag{2.4}$$

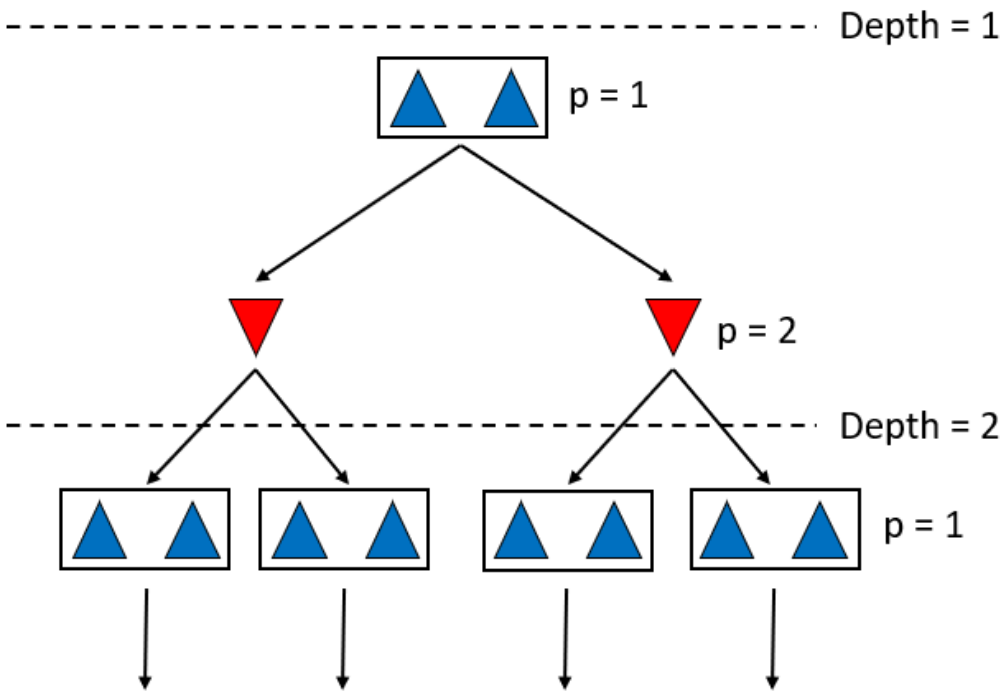

Figure 2.4: GAMA-MCTS framework's tree for a 2 v 1 adversarial environment with simplified layer width

2.2 SPLIT ACTION MULTI-AGENT MONTE CARLO TREE SEARCH

Our second framework is known as Split Action Multi-Agent Monte Carlo Tree Search (SAMA-MCTS). As the name suggests, this framework splits up the teams and considers each node in a specific layer of the tree as an individual player. Therefore, the possible values of p are $1, 2, ..., m+$

n. To search t into the tree, we would need to look $len(p)$ layers deep. For each p_i, the possible actions are DA_i. Therefore, the branching factor for the agents and adversaries is just 9. This framework does not require the duration of all π_i be the same value t. In addition, it can start exploiting for promising actions after trying all of just one of the agents' or adversaries' actions. The only downside is that when it comes time to determine actions for execution, the agents closer to the root of the tree will have higher quality actions due to them being tried more.

A greedy approach is used to determine actions for execution for all m agents by computing Equation 2.4 on the top m layers of the tree. Since this framework relies on m layers of the tree existing, it may not be able to predict an optimal action for each agent if the height of the search tree is less than m. Assuming that the height of the tree is h, where $h < m$, the first h agents' actions are determined, and the last $m - h$ agents' actions are set to a default action where $\pi_{h+1 \to m} = (\text{go straight}, \text{no throttle})$. A simple example of this framework's tree being built for a multi-agent adversarial environment with 2 agents and 1 adversary is shown in Figure 2.5. The true layer width is omitted to save space.

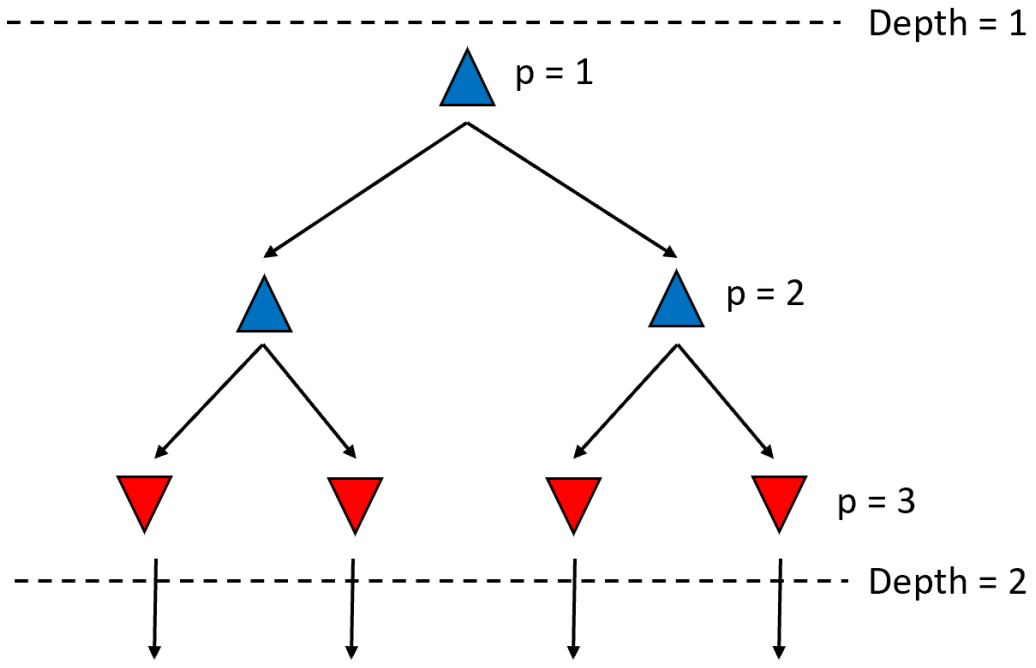

Figure 2.5: SAMA-MCTS framework's tree for a 2 v 1 adversarial environment with simplified layer width

CHAPTER 3

ROOT PARALLELIZATION WITH VARYING AGENT ORDERING

As stated previously, SAMA-MCTS displays some important advantages over GAMA-MCTS. In particular, SAMA-MCTS allows for variable-duration actions and earlier exploitation. By separating out the players into individual layers and keeping track of their assigned team via a flag, the branching factor can be kept constant. There are some downsides, though: agents closer to the root of the tree will have higher quality actions and the tree must be searched to a height of at least m. If the height of the tree is less than m due to factors such as a short timeout, a default action will be chosen for agents between the tree height and m within the agent ordering. This limits the capability of SAMA-MCTS at low timeouts.

One way to increase the the performance of MCTS is to increase the total number of *Simulation* procedures run over the same timeout. With this, a larger amount of the action space can be explored from initial states $\vec{A}$ and $\vec{B}$. The importance of a higher simulation count only grows when adding more players to the game. As the number of agents or adversaries increases, the height required to get to the same point in the tree increases linearly. For example, to fully search s timesteps into the tree, the tree must have a height of $m * s + n * s$.

To achieve a greater amount of simulations, MCTS root parallelization techniques have been used. If N parallel root trees are placed on separate processors and result in similar amounts of simulations, then MCTS should be able to complete N times the number of simulations as compared to the non-parallelization set-up. This is a great way to increase the simulation count, but if all the trees are set-up identically, the downside of needing to search to a height of at least m still remains. When utilizing the vanilla MCTS root parallelization technique with short timeouts, there would be N parallel root trees, each with heights less than m. The information gain by leveraging root parallelization would be limited. To address this issue, we propose our own novel technique known as Root Parallelization with Varying Agent Ordering (RP-VAO).

RP-VAO ensures that our SAMA-MCTS framework never results in any default actions. To do

so, it varies the ordering of the agents between parallel root trees, leading to less overlap and more information gain. There are two main requirements for the parallel root trees: each agent must be ordered first in at least one of the parallel root trees and all the agents must be ordered before all the adversaries. The former imposes that $N \geq m$. The latter biases our search in the direction of the agents which is whom we are determining predicted optimal actions for. If $N = m$, each parallel root tree can be considered the decision-maker for one of the agents. We can look at the first layer of each tree to determine the predicted optimal action for each agent. If $N > m$, there are trees with overlap in the first layer. We combine these overlapping trees when it comes time to determine an action for execution by using an aggregation function. We examined two aggregation functions: best average reward and majority voting. Best average reward takes the average of the rewards from each tree by agent action and selects the maximum for execution. Majority voting determines an action from each tree and assigns it as a vote. In majority voting, the one with the most votes is selected for execution.

Outside of the two main requirements listed above, the remainder of the ordering can be random. For our results, we used the best average reward aggregation function. The computer used for running our experiments was limited to 12 processors. Due to this limitation, there was not much value in the majority voting approach when dealing with a large number of agents. For example, in the case of a 6 v 3 experiment, each agent would have only 2 votes to be used to determine an action for execution. An example of the RP-VAO method for a multi-agent adversarial environment with 2 agents and 1 adversary utilizing 2 processors is shown in Figure 3.1.

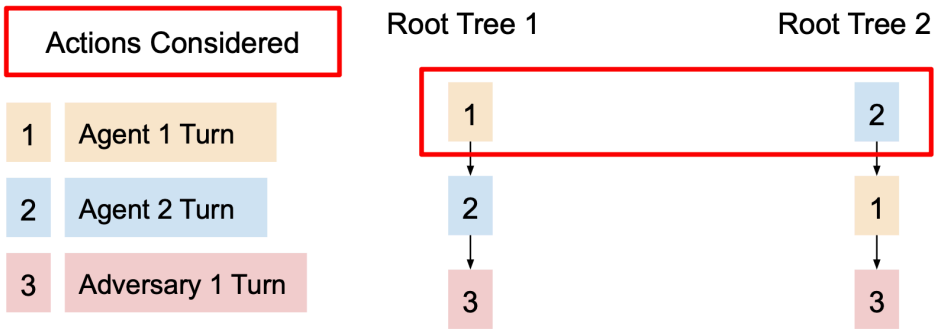

Figure 3.1: RP-VAO method for a 2 v 1 adversarial environment utilizing 2 processors

CHAPTER 4

IMPROVED ACTION SETS

So far, we have discussed ways to improve MCTS by modifying the framework, increasing the number of simulations, and getting "deeper" into the tree by varying the agent ordering. Another way to improve MCTS is to improve the tree's options. In our case, this involves improving on the default action set. To do this, we include closed-loop actions. Closed-loop actions incorporate feedback to make corrections and reduce the error between a reference input and our output. Closed-loop actions are well-aligned with our existing action set because our existing ("open-loop") actions already use discrete time forward predictions. Therefore, closed-loop actions can be predicted in an identical manner to "open-loop" ones.

In the planar 2D multi-agent adversarial environment, the reference input is a target distance and target heading from the current player to an opposing player. Since we are working with multi-agent adversarial environments, we need a way to determine which opposing player to determine the reference input from. We double the size of the cone of fire and use a greedy approach, selecting the closest opposing player in this expanded region. If there is no opposing player in this expanded region, the closest opposing player overall is selected.

The closed-loop action that we have added into the action set is a proportional controller that points the nose of the current player in the direction of the selected opposing player while attempting to maintain a specified separation distance. We only add this closed-loop action into the agents' action set. As will be discussed in the Experiments section of this thesis, the adversaries leverage baseline policies that are identical to this proportional controller. If we added this proportional controller into the adversaries' action set, we would have full knowledge of our adversaries' capabilities while solving MCTS. This would mean that the agents would be able to predict the adversary perfectly. We view the inclusion of this action as something that would limit the broader impact of this work. Therefore, we have excluded it. The adversaries can only pull from the default action set while building the MCTS tree. The agents believe that this is the full extent of the adversaries'

capabilities, allowing for a fair comparison to other experiments.

Further details of the proportional controller are shown in Equations 4.1 and 4.2 where θ_a and e are taken from Figure 2.2. For our results, $\theta_{a,target} = 0°$ and $e_{\text{target}} = 1$ m. This is an advantageous position that places the adversary in the middle of the current agent's cone of fire. The gains used for our proportional controller are $k_{RD} = 1$ and $k_T = 1$. Also, the value of RD is clipped to be between turn left and turn right, and the value of T is clipped to be between no throttle and high throttle.

$$RD = k_{RD}(\theta_a - \theta_{a,target}) \tag{4.1}$$

$$T = k_T(e - e_{\text{target}}) \tag{4.2}$$

This proportional controller is included as a possible action for the agents in both the *Expansion* and *Simulation* procedures. There are now 10 actions in an agent's DA. It is possible to include this closed-loop action because each $node$ contains the current states, $\vec{A}$ and $\vec{B}$. Our action set now has an action that exhibits offensive strategy. In addition, we are introducing an action that is adaptable and responsive.

Now that we have added an extra action into the action set, there are more possible actions to consider during each *Expansion* procedure. Therefore, to get to a similar height in the tree, more simulations are required. Since the agents are lower-performing than the adversaries in our multi-agent adversarial environment, the actions in the agent's DA that contain $RD =$ go straight will only be used in offensive situations. For a separate experiment, we decided to replace $(\text{go straight, no throttle})$, $(\text{go straight, medium throttle})$, and $(\text{go straight, high throttle})$ with our proportional controller. Therefore, there are now 7 actions in an agent's DA. The proportional controller introduces an offensive strategy, and the remaining 6 actions incorporate turns that provide agents with the ability to maneuver out of an adversary's cone of fire. These experiments will allow us to determine how quality and size of an action set affect MCTS performance.

CHAPTER 5

EXPERIMENTS

The frameworks and associated methods presented in the preceding chapters are tested in multi-agent adversarial environments where the team of agents has an advantage in number of players but a disadvantage in player performance. Therefore, the team of agents must effectively select strategic, coordinated actions through teaming to overcome a performance gap. This is done innately through MCTS. Our environments extend the single-agent, planar 2D adversarial environment found in [24] to the multi-agent domain. This prior work looked at the 1 v 1 scenario where both teams had equivalent player performance. Instead, we look at the 2 v 1, 4 v 2, and 6 v 3 scenarios. In each of these scenarios, the agents have a 2 to 1 player advantage, but the players are individually inferior (modeled using added mass). By running these scenarios, we are able to explore the importance of teaming through MCTS and how it can help overcome environment complexity. In addition, we are able to explore ways to overcome MCTS scalability issues that occur when increasing the number of players.

In our scenarios, each player starts with a health score of 200. The team of agents and team of adversaries engage until all the players on one team are eliminated. Eliminations occur when a player reaches a health score of 0. To determine if a player's health score should change, we utilize a variation of the discrete damage reward found in Equation 2.1. A player does a damage of 1 to an opposing player that is in its cone of fire. To make this environment more practical, we impose a limitation where each player can only damage one player on the opposing team during each timestep. This is done by utilizing a greedy approach: if multiple players from the opposing team are in a player's cone of fire, only the opposing player that is closest loses health. A player's loss in health is equivalent to its damage taken.

5.1 DYNAMICS

We utilize exact environment dynamics to those used in [24]. Similar to this prior work, our agent states, $\vec{A}$, and adversary states, $\vec{B}$, are split into position, yaw angle, linear velocity, and yaw rate. Therefore, we are able to use identical equations for lateral force, drag force, moment, and thrust. Equations 5.1, 5.2, 5.3, 5.4, and 5.5 are pulled directly from this prior work and placed here for convenience. S is the side slip angle and $\dot{\psi}$ is the yaw rate. RD and T are our control inputs. C and M are constants that describe how the forces and moments vary based on the previously defined variables. Similar to this prior work, our players are considered simple rigid bodies with constant cross-sectional area (A), moment of inertia, and fluid density (ρ).

$$C_L = C_{L_S} S \tag{5.1}$$

$$\text{Lateral Force} = \rho V^2 A C_L \tag{5.2}$$

$$\text{Drag Force} = \rho V^2 A (C_{D_0} + C_L^2) \tag{5.3}$$

$$\text{Moment} = \rho V^2 A (C_{M_S} S + C_{M_{RD}} RD - C_{M_{\dot{\psi}}} \dot{\psi}) - M_{\dot{\psi}} \dot{\psi} \tag{5.4}$$

$$\text{Thrust} = \text{Thrust}_{\text{max}} T \tag{5.5}$$

Where this thesis differs from this prior work is in player performance. Our team of agents are lower-performing than the team of adversaries. To accomplish this, we vary the mass of the players. Specifically, an individual agent has a mass of 1 kg while an individual adversary has a mass of 0.2857 kg. We assume that the difference in loading occurs at the center of mass, leading to equivalent moment of inertia for the agents and adversaries. All other parameters are kept constant between the agents and adversaries as well. They are pulled directly from the prior work.

It was found that an individual agent had a maximum velocity of 2 m/s, a maximum turn rate of 0.5 rad/s, and a minimum turn radius of 1.97 m. In comparison, an individual adversary had a maximum velocity of 2 m/s, a maximum turn rate of 1.21875 rad/s, and a minimum turn radius of 0.985 m. The adversaries have much higher maneuverability. When comparing, an individual

adversary has half the minimum turn radius, meaning it has 2x the performance. Therefore, an advantage in numbers is not guaranteed to provide a victory.

5.2 TESTING SUITES

We set up 3 different testing suites known as clock trials. Each set of clock trials represents one of the 2 v 1, 4 v 2, or 6 v 3 scenarios discussed earlier. These clock trials test different starting locations for the team of adversaries. In doing so, we are able to test positions that are advantageous, neutral, and disadvantageous for the team of agents. Our clock trials include 16 different combinations of starting locations. We run 10 trials per combination. MCTS does not return the same actions for execution during each run due to randomness in the rollout policy, exploration vs. exploitation characteristics, and a short timeout. Therefore, running multiple trials per combination is essential to showing the true effects of our frameworks and associated methods.

Figures 5.1, 5.2, and 5.3 shows the set-ups for our 2 v 1, 4 v 2, and 6 v 3 clock trials, respectively. Each set of adversary or red positions represents a different starting location. Each team starts in a common flying formation based on the number of players. For example, when there are 4 players, the team starts in the finger-four formation as shown by each set of agent or blue positions in the 4 v 2 clock trials.

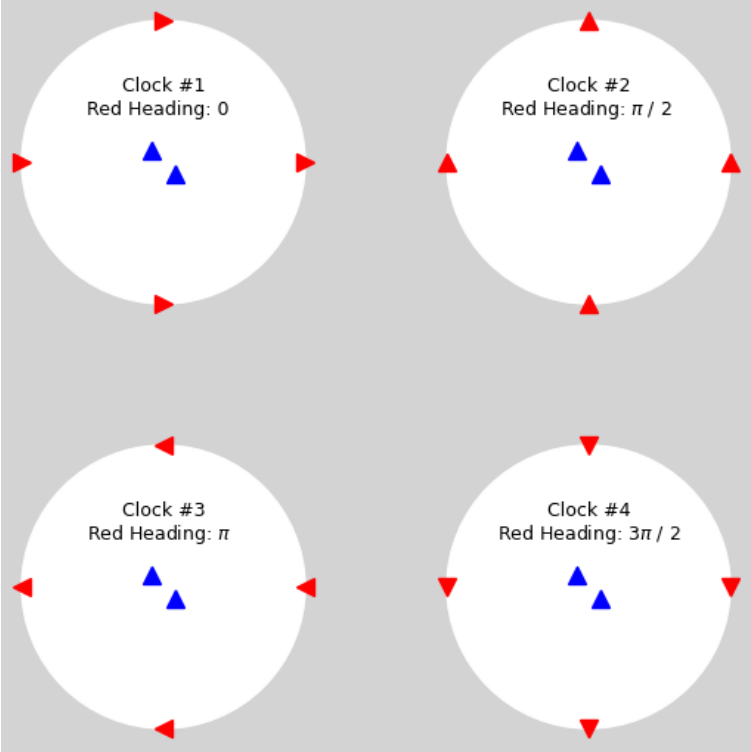

Figure 5.1: 2 v 1 clock trials

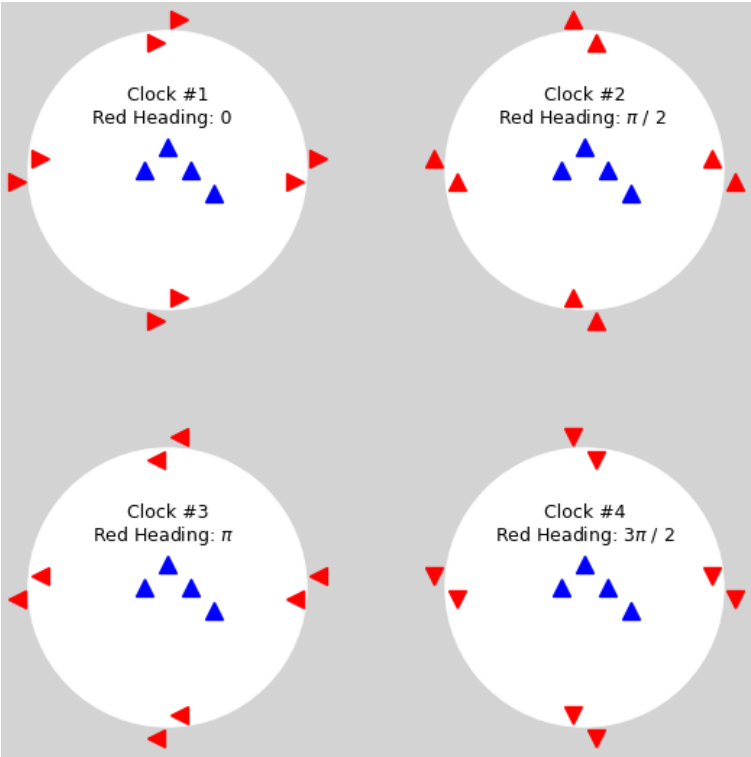

Figure 5.2: 4 v 2 clock trials

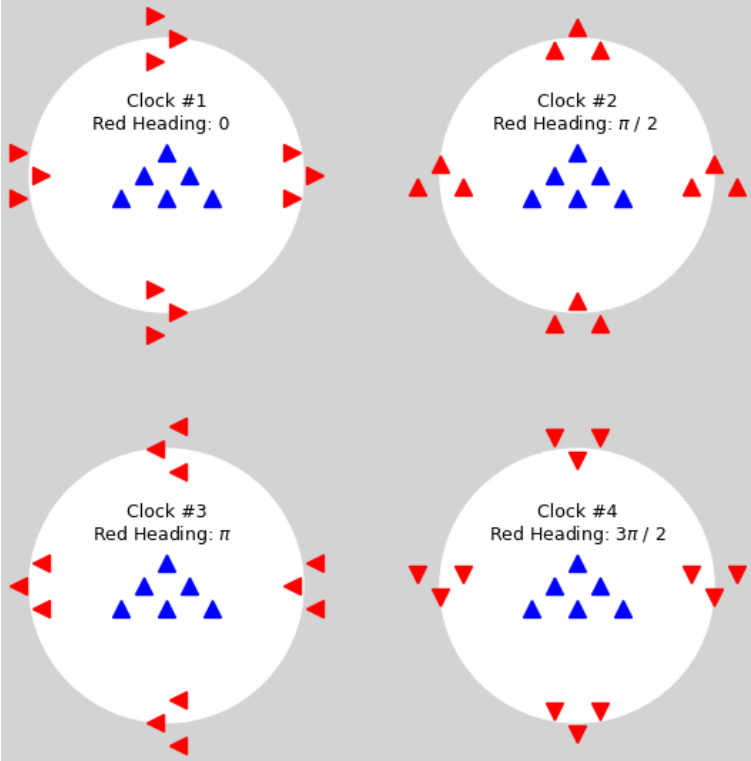

Figure 5.3: 6 v 3 clock trials

From each starting location combination, we let the team of agents running our MCTS frameworks and associated methods engage the team of higher-performing adversaries running a baseline policy. This baseline policy is identical to the proportional controller displayed in Equations 4.1 and 4.2. The only difference is the replacement of θ_a with θ_b. The adversaries do not demonstrate means of intelligence and are not meant to be incredibly challenging. Instead, they provide us with a benchmark for comparison between experiments.

The engagement ends when one full team is eliminated. If all the adversaries are eliminated, we consider this a win for our MCTS frameworks and associated methods. On the other hand, if all the agents are eliminated, we consider this a loss as the agents were not able to overcome the higher-performing baseline adversaries via teaming. We track this metric for each of the 16 different starting combinations within the clock trial. We take the average of the win percentages for each clock trial starting combination to get an overall win percentage score for the set-up. This value is used for comparison. In addition, we track the average number of simulations for each set-up. With

more simulations, MCTS should perform better since the basis of the decision making process is the outcome of these simulations. Therefore, this is an important metric to track. When all other factors are considered equal, actions determined from MCTS with a larger simulation count should be of higher quality.

The results of MCTS are highly dependent on the code's runtime and computer architecture. Our frameworks and associated methods are built with Python and run on an Intel i7-12700 processor using one thread. To improve the results, the code can be further optimized or more computational power can be utilized. This would shift our win percentages and simulation counts up but would not change the overall takeaways of the thesis.

CHAPTER 6

RESULTS AND DISCUSSION

6.1 BASELINE

To stress the difficulty of our problem, we started by running a 2 v 1 clock trial where both the agents and the higher-performing adversaries follow the baseline policy. In this scenario, the agents have a player advantage, but the players are individually inferior. Since we were utilizing the baseline policy instead of MCTS, there was no teaming component. The average win percentage over the 16 different starting combinations (10 trials each for a total of 160 trials) was a mere 25%. This means that the advantage in numbers generally did not lead to victories. These results illustrate that the need for intelligent teaming strategies.

These results act as a benchmark for future experiments. We expected that by allowing our agents to exhibit intelligence through teaming via MCTS, average win percentage would increase.

6.2 FRAMEWORK COMPARISON

After running the baseline experiment, we shifted our focus to testing both of our multi-agent MCTS frameworks. We ran multiple tests to determine which framework performed best under different conditions. We started with running GAMA-MCTS for the 2 v 1, 4 v 2, and 6 v 3 clock trials with a 5 s timeout. We then performed the same tests for SAMA-MCTS. We kept track of three metrics for comparison: win percentage, average simulations, and average time to solve. As stated previously, we wanted to maximize win percentage and average simulations. Average time to solve is the average runtime of MCTS for determining an action for execution. The closer the average time to solve to the timeout, the better. If the average time to solve is much larger than the timeout, there is overhead caused by our framework set-up, and we are not adhering to the algorithm requirements.

Tables 6.1 and 6.2 display these results. For our results at 5 s timeout, we used multi-processing and ran all four clocks in unison. This helped limit the overall testing suite runtime. Increasing the

number of processes has an adverse effect on the average simulations value due to increased CPU utilization. These results are further summarized in Figures 6.1, 6.2, and 6.3.

Table 6.1: GAMA-MCTS results for a 5 s timeout, running each clock in unison

Clock Trial	Win Percentage	Average Simulations	Average Time to Solve
2 v 1	86.25%	990.88	5.0028 s
4 v 2	69.38%	553.62	5.0098 s
6 v 3	55.62%	287.31	5.056 s

Table 6.2: SAMA-MCTS results for a 5 s timeout, running each clock in unison

Clock Trial	Win Percentage	Average Simulations	Average Time to Solve
2 v 1	89.38%	1002.31	5.0026 s
4 v 2	82.5%	556.81	5.0057 s
6 v 3	56.25%	459.06	5.0069 s

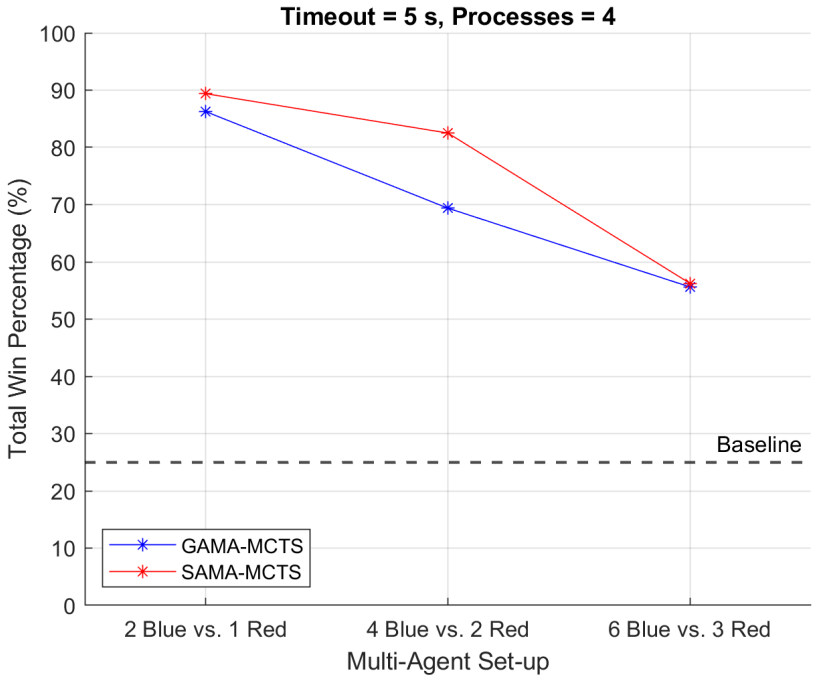

Figure 6.1: Framework win percentage comparison for a 5 s timeout, running each clock in unison

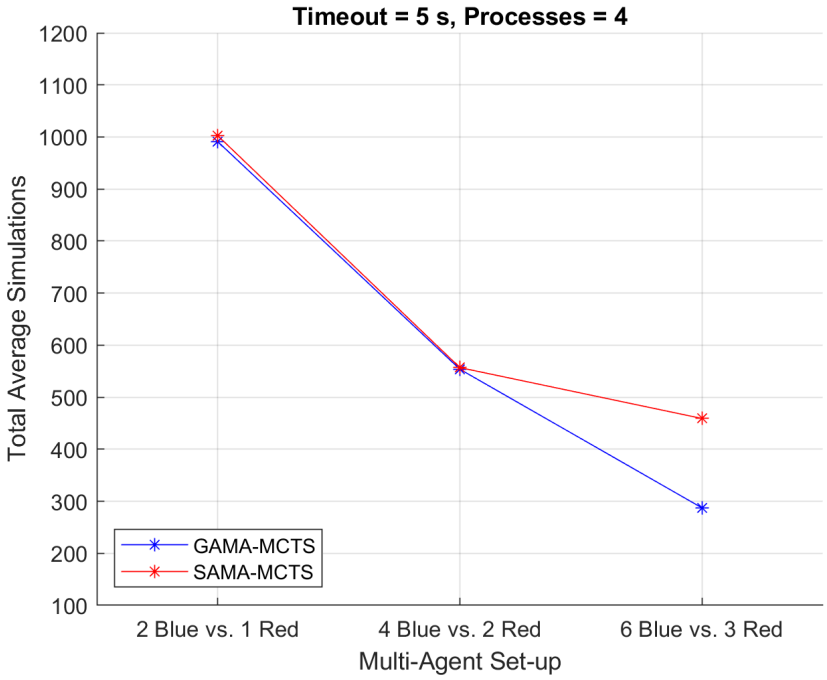

Figure 6.2: Framework average simulations comparison for a 5 s timeout, running each clock in unison

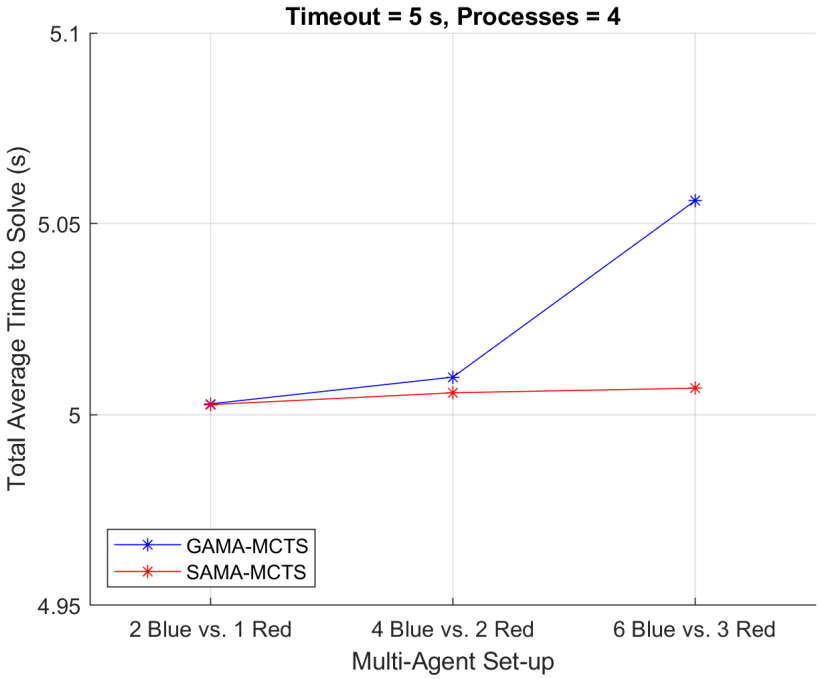

Figure 6.3: Framework average time to solve comparison for a 5 s timeout, running each clock in unison

We then ran these same tests for a 1 s timeout. Tables 6.3 and 6.4 display these results. For our results at 1 s timeout, no multi-processing was used. These results are further summarized in Figures 6.4, 6.5, and 6.6.

Table 6.3: GAMA-MCTS results for a 1 s timeout

Clock Trial	Win Percentage	Average Simulations	Average Time to Solve
2 v 1	60.62%	157.56	1.0067 s
4 v 2	45.62%	108.88	1.0143 s
6 v 3	51.25%	41.44	1.3441 s

Table 6.4: SAMA-MCTS results for a 1 s timeout

Clock Trial	Win Percentage	Average Simulations	Average Time to Solve
2 v 1	54.38%	175.12	1.0052 s
4 v 2	33.12%	123.75	1.0051 s
6 v 3	23.12%	92.56	1.0063 s

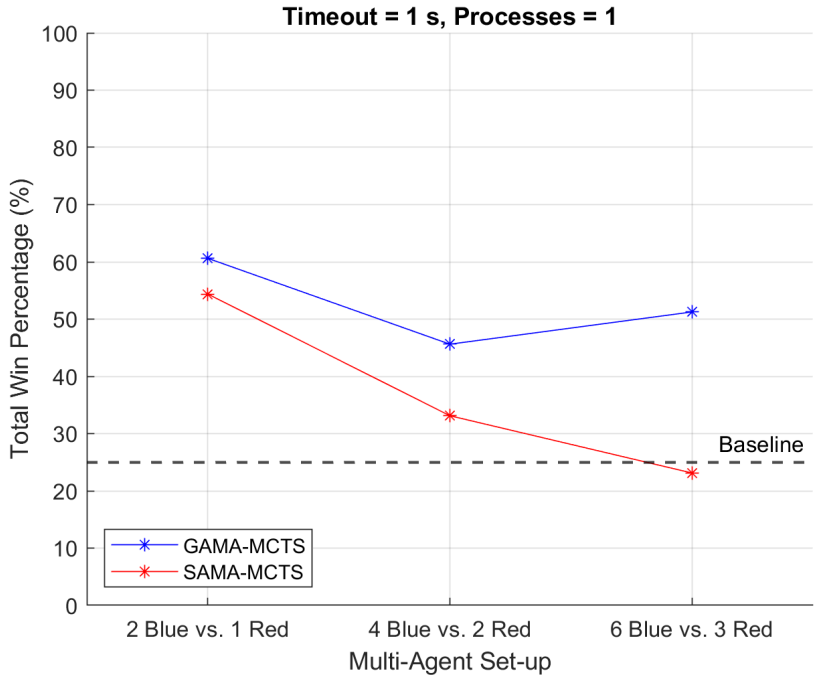

Figure 6.4: Framework win percentage comparison for a 1 s timeout

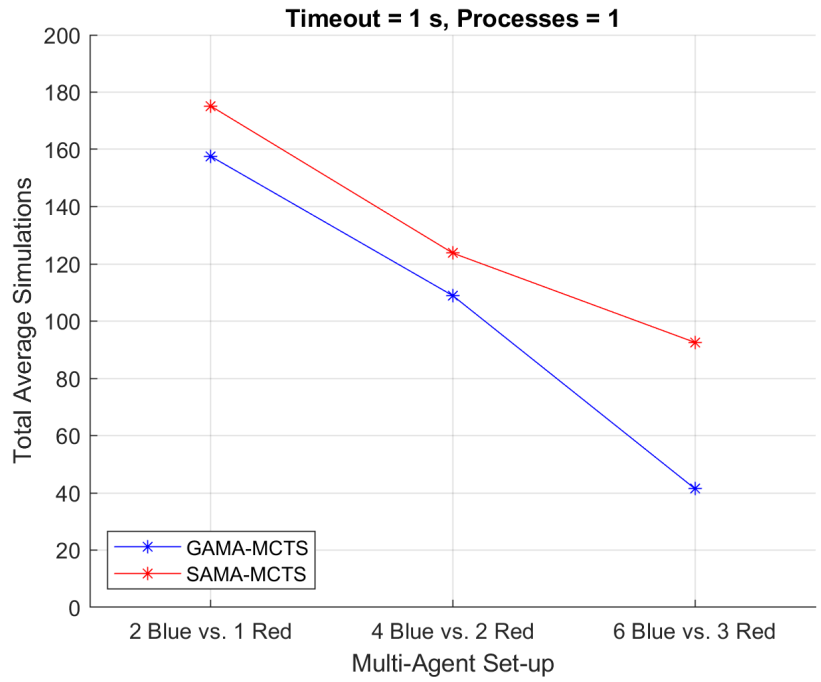

Figure 6.5: Framework average simulations comparison for a 1 s timeout

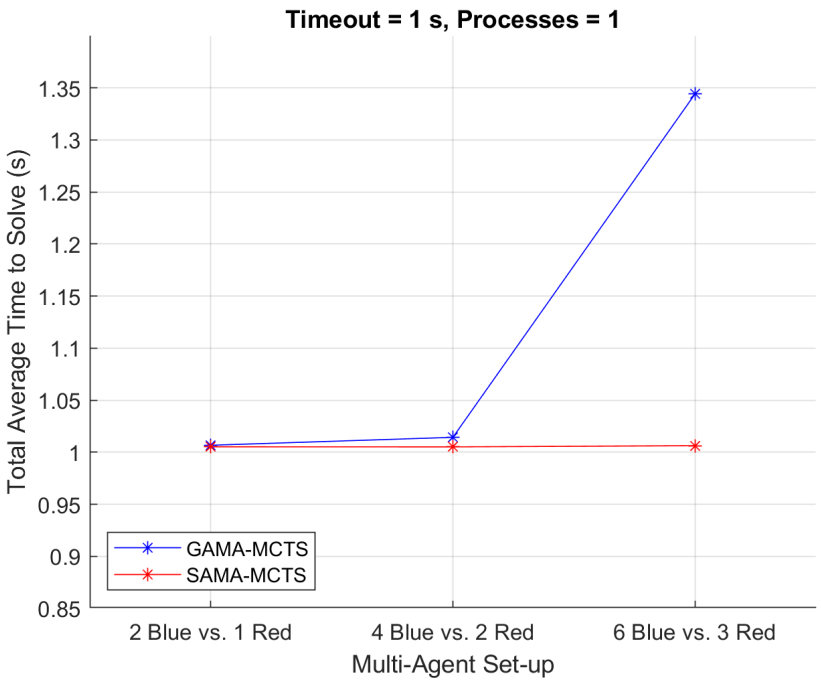

Figure 6.6: Framework average time to solve comparison for a 1 s timeout

For the 2 v 1 clock trials, we see a large increase in win percentage over the baseline at both

timeouts. At a 5 s timeout, both frameworks see decreased win percentage, decreased average simulations, and increased average time to solve as the number of players increases. This makes sense, when comparing the 6 v 3 scenario with the 2 v 1 scenario, there are 6 more players in the tree that need to have their actions forward-simulated. SAMA-MCTS outperforms GAMA-MCTS in every metric. GAMA-MCTS struggles more when the number of players increases. In particular, GAMA-MCTS has a lower average simulations and worse average time to solve (287.31 and 5.056 s, respectively) compared to SAMA-MCTS (459.06 and 5.0069 s, repectively) in the 6 v 3 clock trials.

For the 1 s timeout, the results are considerably worse than those for the 5 s timeout. In addition, there are cases where GAMA-MCTS outperforms SAMA-MCTS. With less time allocated to the decision making process, we get a lower number of simulations and lower quality action determination. Surprisingly, we see that GAMA-MCTS outperforms SAMA-MCTS in win percentage. Though, GAMA-MCTS continues to perform worse in average simulations and average time to solve. This is due to GAMA-MCTS' high branching factor. In the *Expansion* procedure, the tree is expanded to include all possible actions for the leaf node's child. For GAMA-MCTS in the 6 v 3 scenario, the branching factor for the agents is $9^6 = 531,441$. Therefore, there are $531,441$ possible actions or newly added nodes to the tree that need to call *Simulation*. This leads to a large amount of overhead as the average time to solve for GAMA-MCTS in this scenario is 1.3441 s. Therefore, an extra 0.3441 s is being allocated to decision making. In a critical environment where each decision needs to occur in a timely manner, this is unacceptable. This occurs because we only check the timeout before calling *Selection*.

On the other hand, SAMA-MCTS maintains good average time to solves, only increasing to 1.0063 s for the 6 v 3 scenario. Due to how the tree is built, the branching factor for SAMA-MCTS is only 9. The win percentages decrease from 54.38% for the 2 v 1 scenario to their lowest point of 23.12% for the 6 v 3 scenario. When determining actions for a large amount of agents at a low timeout, it is likely that many of the agents are receiving default actions because the height of the tree is small. This leads to low winning percentages.

The results from the 5 s timeout illustrate the improvements achieved by SAMA-MCTS. These results led us to use SAMA-MCTS for the rest of this thesis. In the next section, we present results for RP-VAO which addresses the default action issue that occurs at low timeouts and for a large amounts of players.

6.3 RP-VAO

As stated previously, our RP-VAO method requires that $N \geq m$. Therefore, in our 6 v 3 scenario, we need at least 6 parallel root trees in order to use RP-VAO. In this section, we present results for each of our testing suites utilizing 6, 9, and 12 parallel root trees at a timeout of 1 s. We focused on low timeouts for this section because this is where we saw performance reductions for SAMA-MCTS. Utilizing RP-VAO for larger timeouts, such as 5 s, results in near 100% win percentage for all scenarios due to the substantial amount of simulations that can be performed in the larger amount of time. Though, the same general trends for average simulations and average time to solve remain (except they are about 5 times greater in magnitude).

Table 6.5 displays the results for our 2 v 1 scenario. Similar to previous 1 s timeout results, each clock is run one at a time. The row for the parallel root trees value of 1 is the result from the above section that did not use RP-VAO. Results for the 4 v 2 and 6 v 3 scenarios are shown in Tables 6.6 and 6.7, respectively. These results are further summarized in Figures 6.7, 6.8, and 6.9.

Table 6.5: RP-VAO results for the 2 v 1 scenario at 1 s timeout

Parallel Root Trees	**Win Percentage**	**Average Simulations**	**Average Time to Solve**
1	54.38%	157.56	1.0067 s
6	76.88%	933.69	1.111 s
9	71.25%	1228.94	1.2186 s
12	76.25%	1303.19	1.1331 s

Table 6.6: RP-VAO results for the 4 v 2 scenario at 1 s timeout

Parallel Root Trees	Win Percentage	Average Simulations	Average Time to Solve
1	33.12%	123.75	1.0051 s
6	87.5%	653.44	1.0591 s
9	93.12%	836.75	1.0767 s
12	86.25%	895.88	1.0885 s

Table 6.7: RP-VAO results for the 6 v 3 scenario at 1 s timeout

Parallel Root Trees	Win Percentage	Average Simulations	Average Time to Solve
1	23.12%	92.56	1.0063 s
6	75%	485.19	1.0589 s
9	83.75%	641.75	1.0783 s
12	78.75%	684.56	1.0911 s

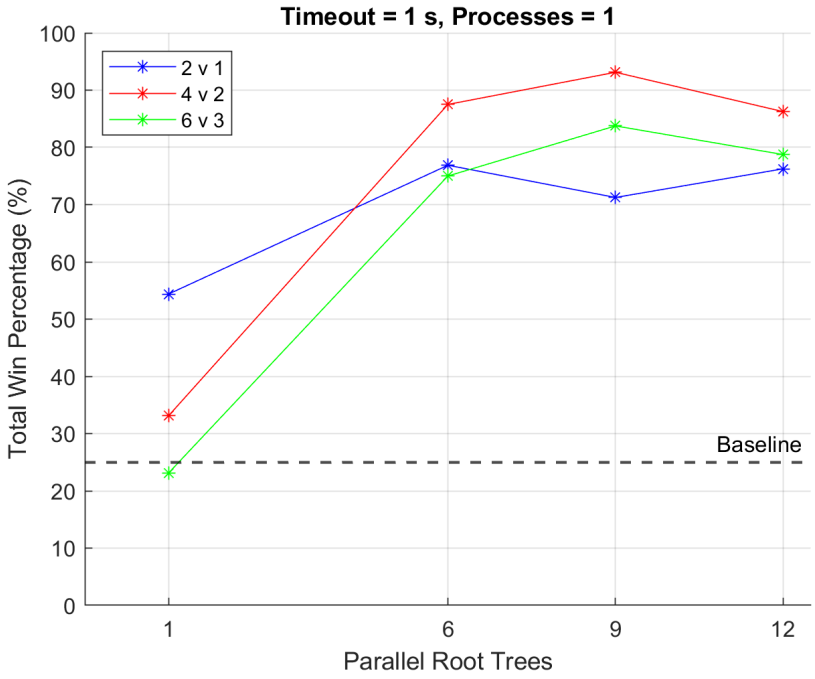

Figure 6.7: RP-VAO win percentage comparison for a 1 s timeout

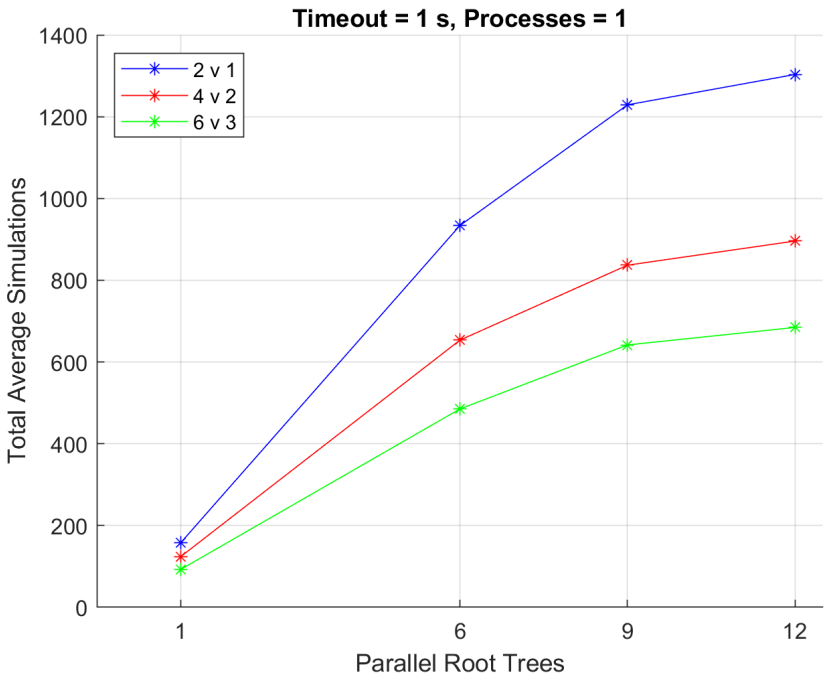

Figure 6.8: RP-VAO average simulations comparison for a 1 s timeout

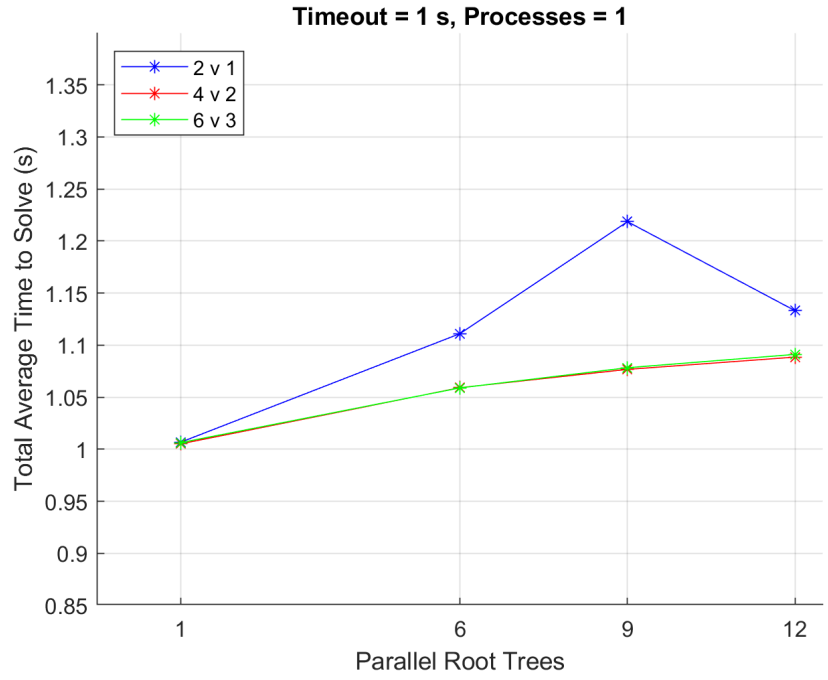

Figure 6.9: RP-VAO average time to solve comparison for a 1 s timeout

Through RP-VAO, we see large increases in win percentage. We see a 22.5%, 60%, and 60.63%

win percentage boost for the 2 v 1, 4 v 2, and 6 v 3 scenarios, respectively. These win percentage boosts are due to the novel ordering technique and the large increase in average simulations brought by RP-VAO. From 1 to 12 parallel root trees, we see an increase in average simulations from 157.56 to 1303.19 for 2 v 1, from 123.75 to 895.88 for 4 v 2, and from 92.56 to 684.56 for 6 v 3. The increases for the more complex environments are smaller because there are more players that need to have their actions forward-simulated. In addition, the increase in average simulations for each scenario is not linear. Instead, they exhibit asymptotic behavior as shown by the increase in average simulations from 9 to 12 parallel root trees being small. This means we are reaching the limits of our computational capability (100% CPU utilization). There is also more overhead required to instantiate more parallel root trees seen by the increasing average time to solves which could be contributing to the trends for average simulations.

The larger number of parallel root trees do not necessarily result in best performance. For example, the best results for 2 v 1, 4 v 2, and 6 v 3 occur at 6, 9, and 9 parallel root trees, respectively. This could be due to unnecessary overlap of information occurring at larger amounts of parallel root trees or due to randomness in the *Simulation* procedure. All in all, RP-VAO is a promising technique used to overcome scalability issues in our SAMA-MCTS framework that arise when increasing the number of players.

6.4 IMPROVED ACTION SETS

The final results that we present are for our improved action sets. To show the benefits of adding the offensive strategy action, we ran these tests at a timeout of 5 s. This allowed us to quantify the difference between experiments without dealing with the limitations caused by low timeouts. For simplicity in this section, we call the agent's DA with 10 actions the combo action set and the agent's DA with 7 actions the compact action set. RP-VAO is not used in this section.

Tables 6.8, 6.9, and 6.10 display the results for our 2 v 1, 4 v 2, and 6 v 3 scenarios, respectively. The rows for the default action set are taken directly from Table 6.2. Similar to previous 5 s timeout results, we used multi-processing and ran all four clocks in unison. The win percentage results are

further summarized in Figure 6.10. Figures summarizing the average simulations and average time to solve results for these experiments are omitted since these values remain relatively constant.

Table 6.8: Improved action sets results for the 2 v 1 scenario at 5 s timeout, running each clock in unison

Action Set	Win Percentage	Average Simulations	Average Time to Solve
Default	89.38%	1002.31	5.0026 s
Combo	93.75%	962.88	5.0025 s
Compact	96.25%	1091.69	5.0024 s

Table 6.9: Improved action sets results for the 4 v 2 scenario at 5 s timeout, running each clock in unison

Action Set	Win Percentage	Average Simulations	Average Time to Solve
Default	82.5%	556.81	5.0057 s
Combo	89.38%	517.94	5.0064 s
Compact	99.38%	490.19	5.0064 s

Table 6.10: Improved action sets results for the 6 v 3 scenario at 5 s timeout, running each clock in unison

Action Set	Win Percentage	Average Simulations	Average Time to Solve
Default	56.25%	459.06	5.0069 s
Combo	65%	420.31	5.0077 s
Compact	89.38%	394.62	5.0074 s

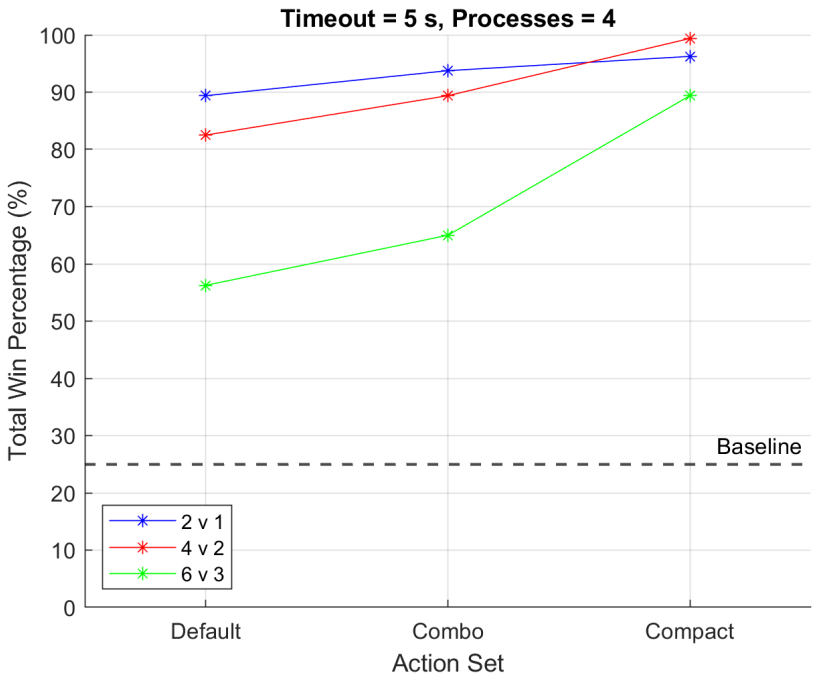

Figure 6.10: Improved action sets win percentage comparison for a 5 s timeout, running each clock in unison

As we move from the default action set to the combo action set, we see increased winning percentage for all scenarios. Specifically, we see increases of 4.37%, 6.88%, and 8.75% for the 2 v 1, 4 v 2, and 6 v 3 scenarios, respectively. Therefore, adding the offensive strategy action, represented by a proportional controller, does benefit the team of agents. Though, the win percentage for the 6 v 3 scenario is still low, only reaching 65%. By throwing away the $RD =$ go straight actions, thus decreasing the number of possible actions from 10 to 7, we can get further in the tree with the same number of simulations. This is beneficial when the number of players in the game increases.

Due to this, we saw further improvements by moving to the compact action set. We reached 96.25%, 99.38%, and 89.38% for the 2 v 1, 4 v 2, and 6 v 3 scenarios, respectively. By limiting our action set to only include those that were essential, we are able to get deeper in the tree quicker. This is an important trade-off that occurs in MCTS for adversarial environments. We want action sets that are as small as possible while still effectively solving the environment. If the action set is too rich, it would take too long to build the tree. Conversely, if the action set is too limited, it will not be able to solve the environment.

CHAPTER 7

LIMITATIONS AND FUTURE WORK

As stated previously, MCTS is highly dependent on the code's runtime and computer architecture. Therefore, our results are limited in this regard. To improve our results further, we could continue to optimize our code or purchase more compute. In addition, we could move from Python to C++. Python is an interpreted language while C++ is a compiled language, meaning that C++ has much higher execution efficiency.

Our results are also limited by the use of random rollouts in the *Simulation* procedure. Each time we run MCTS, we could receive a different action for execution. Eventually, if we perform enough simulations, this returned action should converge, but we are running our experiments at relatively low timeouts to simulate real-time behavior. [17] gets around this by incorporating two neural networks, a policy network and a value network. The value network replaces the random rollouts directly, providing an immediate evaluation from the current leaf node. By removing the random rollouts, efficiency is massively increased.

In terms of future work, these frameworks and associated methods should be applied to more complicated environments than the planar 2D multi-agent adversarial environment. This could be tested by developing a more complex dynamics model than the one pulled from [24]. Additionally, we could attempt to use our frameworks and associated methods in non-adversarial multi-agent environments to see if they improve on the current literature. For strategic actions, we could explore other closed-loop or feedback actions to insert into the action set. Specifically, we could add actions that incorporate defensive strategy or add a trained Deep RL model that maps current states to agent actions. Finally, we could look at ways to bypass the use of forward-simulated actions. For example, [24] made use of the MA.

CHAPTER 8

CONCLUSION

Our frameworks and associated methods led to large increases in performance for the multi-agent adversarial environment where a team of agents leveraging MCTS engage a team of higher-performing baseline adversaries. In the baseline experiment where there is lack of intelligence, we saw a low winning percentage of 25% for the 2 v 1 clock trials. We looked to improve on this baseline in all future experiments. We presented two multi-agent MCTS frameworks, GAMA-MCTS and SAMA-MCTS. It was determined that SAMA-MCTS was the superior framework for multi-agent adversarial environments due to its low branching factor. Based on how the tree is set-up, we were able to start exploiting its promising actions sooner. For a 5 s timeout, we received win percentages of 89.38%, 82.5%, and 56.25% for the 2 v 1, 4 v 2, and 6 v 3 scenarios, respectively. These values decreased to 54.38%, 33.12%, and 23.12% for a timeout of 1 s. Either way, there was a large increase in performance for the 2 v 1 clock trials over our baseline experiment.

The decrease in win percentage as we increase the number of players or lower the timeout can be attributed to a lack of height in the tree while utilizing SAMA-MCTS. This led to default actions selected for execution. RP-VAO was introduced to overcome this. To challenge our method, we utilized RP-VAO for experiments with a 1 s timeout. We saw increases of 54.38% to 76.88%, 33.12% to 93.12%, and 23.12% to 83.75% for the 2 v 1, 4 v 2, and 6 v 3 scenarios, respectively. At a 5 s timeout, these win percentages all jumped to nearly 100% due to the large amounts of simulations that could be achieved via RP-VAO at the larger timeout. Therefore, RP-VAO did an exceptional job in overcoming the scalability issues found in SAMA-MCTS.

Finally, we showed that utilizing improved action sets led to increases in win percentage. Our default action set only contains "open-loop" actions. Each *node* stores the current states, so it is possible to include closed-loop or feedback actions. We included a proportional controller with offensive strategy, increasing our win percentage for the 5 s timeout from 89.38% to 93.75%, 82.5% to 89.38%, and 56.25% to 65% for the 2 v 1, 4 v 2, and 6 v 3 scenarios, respectively. By decreasing

the number of actions to 7 and only including those that were essential, these win percentages increased to 96.25%, 99.38%, and 89.38%. Thus, there is a trade-off between action quality and number of actions that needs to be considered when working with MCTS.

All in all, we effectively extended MCTS to a multi-agent adversarial environment. Our frameworks and associated methods incorporated innate teaming, allowing a team of agents to beat a team of higher-performing baseline adversaries. By leveraging MCTS, no pre-training occurred. Our frameworks and associated methods are relevant for a differing number of players, differing player performance, or a differing adversary strategy, giving it the upper-hand over Deep RL for these types of environments.

REFERENCES

[1] T. P. Lillicrap *et al.*, *Continuous control with deep reinforcement learning*, 2019. arXiv: 1509.02971 [cs.LG].

[2] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, *Proximal policy optimization algorithms*, 2017. arXiv: 1707.06347 [cs.LG].

[3] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, *Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor*, 2018. arXiv: 1801.01290 [cs.LG].

[4] A. P. Pope *et al.*, "Hierarchical reinforcement learning for air-to-air combat," *CoRR*, vol. abs/2105.00990, 2021. arXiv: 2105.00990. [Online]. Available: https://arxiv.org/abs/2105.00990.

[5] A. P. Pope *et al.*, "Hierarchical reinforcement learning for air combat at darpa's alphadogfight trials," *IEEE Transactions on Artificial Intelligence*, vol. 4, no. 6, pp. 1371–1385, 2023. doi: 10.1109/TAI.2022.3222143.

[6] Heron Systems Inc., *Heron systems at darpa alpha dogfight trials*, YouTube. https://www.youtube.com/watch?v=IIdE5XFTA88.

[7] O. Vinyals, I. Babuschkin, W. M. Czarnecki, *et al.*, "Grandmaster level in starcraft ii using multi-agent reinforcement learning," *Nature*, vol. 575, pp. 350–354, 2019. doi: 10.1038/s41586-019-1724-z. [Online]. Available: https://doi.org/10.1038/s41586-019-1724-z.

[8] A. Selmonaj, O. Szehr, G. Del Rio, A. Antonucci, A. Schneider, and M. Rüegsegger, *Hierarchical multi-agent reinforcement learning for air combat maneuvering*, 2023. arXiv: 2309.11247 [cs.LG].

[9] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *Machine Learning: ECML 2006*, J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 282–293, isbn: 978-3-540-46056-5.

[10] R. Coulom, "Efficient selectivity and backup operators in monte-carlo tree search," in *Computers and Games*, H. J. van den Herik, P. Ciancarini, and H. H. L. M. (Donkers, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 72–83, isbn: 978-3-540-75538-8.

[11] M. Świechowski, K. Godlewski, B. Sawicki, and J. Mańdziuk, "Monte carlo tree search: A review of recent modifications and applications," *Artificial Intelligence Review*, vol. 56, no. 3, pp. 2497–2562, Jul. 2022, issn: 1573-7462. doi: 10.1007/s10462-022-10228-y. [Online]. Available: http://dx.doi.org/10.1007/s10462-022-10228-y.

[12] C. B. Browne *et al.*, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012. doi: 10.1109/TCIAIG.2012.2186810.

[13] S. J. Russell, *Artificial intelligence a modern approach*. Pearson Education, Inc., 2010.

[14] G. Chaslot, M. H. M. Winands, and H. J. van den Herik, "Parallel monte-carlo tree search," *Computers and Games*, vol. 5131, no. 3, pp. 60–71, Mar. 2008, Springer, Berlin, Heidelberg, issn: 0278-3649. doi: 10.1007/978-3-540-87608-3_6. [Online]. Available: https://doi.org/10.1007/978-3-540-87608-3_6.

[15] K. Kurzer, C. Hörtnagl, and J. M. Zöllner, "Parallelization of monte carlo tree search in continuous domains," *Computers and Games*, vol. 1, no. 1, p. 1, Mar. 2020, 1, issn: 1. doi: 10.48550/arXiv.2003.13741. [Online]. Available: https://doi.org/10.48550/arXiv.2003.13741.

[16] K. Kurzer, C. Hörtnagl, and J. M. Zöllner, "Decentralized cooperative planning for automated vehicles with continuous monte carlo tree search," *Computers and Games*, vol. 1, no. 1, p. 1, Sep. 2018, 1, issn: 1. doi: 10.48550/arXiv.1809.03200. [Online]. Available: https://doi.org/10.48550/arXiv.1809.03200.

[17] D. Silver *et al.*, "Mastering the Game of Go with Deep Neural Networks and Tree Search," en, *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016, issn: 0028-0836, 1476-4687. doi: 10.1038/nature16961. [Online]. Available: http://www.nature.com/articles/nature16961.

[18] D. Silver *et al.*, "Mastering the Game of Go Without Human Knowledge," en, *Nature*, vol. 550, no. 7676, pp. 354–359, Oct. 2017, issn: 0028-0836, 1476-4687. doi: 10.1038/nature24270. [Online]. Available: http://www.nature.com/articles/nature24270.

[19] D. Silver *et al.*, "A General Reinforcement Learning Algorithm That Masters Chess, Shogi, and Go Through Self-Play," en, *Science*, vol. 362, no. 6419, pp. 1140–1144, Dec. 2018, issn: 0036-8075, 1095-9203. doi: 10.1126/science.aar6404. [Online]. Available: https://www.science.org/doi/10.1126/science.aar6404.

[20] G. Van den Broeck, K. Driessens, and J. Ramon, "Monte-carlo tree search in poker using expected reward distributions," in *Advances in Machine Learning*, Z.-H. Zhou and T. Washio, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 367–381, isbn: 978-3-642-05224-8.

[21] T. Pepels, M. Winands, and M. Lanctot, "Real-time monte-carlo tree search in ms pac-man," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 6, pp. 245–257, Sep. 2014. doi: 10.1109/TCIAIG.2013.2291577.

[22] A. Uriarte and S. Ontañón, "Improving monte carlo tree search policies in starcraft via probabilistic models learned from replay data," *Proceedings of the AAAI Conference on Artificial*

Intelligence and Interactive Digital Entertainment, vol. 12, no. 1, pp. 100–106, Jun. 2021. doi: 10.1609/aiide.v12i1.12852. [Online]. Available: https://ojs.aaai.org/index.php/AIIDE/article/view/12852.

[23] A. Santos, P. A. Santos, and F. S. Melo, "Monte carlo tree search experiments in hearthstone," in *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, Aug. 2017, pp. 272–279. doi: 10.1109/CIG.2017.8080446.

[24] Z. C. Goddard, R. Rajasekar, M. Mocharla, G. Manaster, K. Williams, and A. Mazumdar, "Leveraging machine learning for generating and utilizing motion primitives in adversarial environments," *Journal of Aerospace Information Systems*, vol. 21, no. 2, pp. 127–139, 2024. doi: 10.2514/1.I011283. eprint: https://doi.org/10.2514/1.I011283. [Online]. Available: https://doi.org/10.2514/1.I011283.

[25] E. Frazzoli, M. Dahleh, and E. Feron, "A hybrid control architecture for aggressive maneuvering of autonomous helicopters," in *Proceedings of the 38th IEEE Conference on Decision and Control (Cat. No.99CH36304)*, vol. 3, 1999, 2471–2476 vol.3. doi: 10.1109/CDC.1999.831296.

[26] E. Frazzoli, M. Dahleh, and E. Feron, "Robust hybrid control for autonomous vehicle motion planning," in *Proceedings of the 39th IEEE Conference on Decision and Control (Cat. No.00CH37187)*, vol. 1, 2000, 821–826 vol.1. doi: 10.1109/CDC.2000.912871.

[27] J. S. McGrew, "Real-Time Maneuvering Decisions for Autonomous Air Combat," eng, Accepted: 2009-03-20T19:32:46Z, Thesis, Massachusetts Institute of Technology, 2008. [Online]. Available: https://dspace.mit.edu/handle/1721.1/44927.

[28] J. S. McGrew, J. P. How, B. Williams, and N. Roy, "Air-Combat Strategy Using Approximate Dynamic Programming," en, *Journal of Guidance, Control, and Dynamics*, vol. 33, no. 5, pp. 1641–1654, Sep. 2010, issn: 0731-5090, 1533-3884. doi: 10.2514/1.46815. [Online]. Available: https://arc.aiaa.org/doi/10.2514/1.46815.