5-2020

Coordinated Autonomy of Unmanned Aerial Vehicles (UAVs)

Paribartan Dhakal

The University of Southern Mississippi

Coordinated Autonomy of Unmanned Aerial Vehicles (UAVs)

by

Paribartan Dhakal

A Thesis
Submitted to the Honors College of
The University of Southern Mississippi
in Partial Fulfillment
of Honors Requirements

May 2020

Approved by:

____
Bikramjit Banerjee, Ph.D., Thesis Adviser
Professor of Computer Science

____
Andrew Sung, Ph.D., Director
School of Computing Science and Computer Engineering

____
Ellen Weinauer, Ph.D., Dean
Honors College

Abstract

Unmanned Aerial Vehicles (UAVs) are being extensively used in diverse sectors of the society for various tasks ranging from videography to an extremely sensitive situation such as first responders helping during a disaster. It has been seen that if a fleet of UAVs is deployed, they can perform a task quicker and more efficiently than a single UAV. With an increase in the number of UAVs, a problem arises of handling them with proper control structures. It has been studied that the Behavior Trees (BT) can be a better control architecture to handle the autonomous vehicles, as BTs are more reactive to changes. Being a fixed-rule decision model, there is still some bias on how a BT chooses its behavior. There has been some work to solve this issue by including the Reinforcement Learning (RL) technique for BTs so that agents can better learn their BTs and choose actions without bias. But this work only applies to a single agent system. Hence, we have proposed a solution on how multiple UAVs can apply RL in a cooperative scenario to learn their BTs. We show how the inclusion of communication between agents will impact the way an autonomous agent learns its BT.

Keywords: UAVs, Behavior Trees, Coordinaition, Multi-Agents

Dedication

To mum, dad, Ishika and Elina

Acknowledgements

I would like to express my sincerest gratitude to my thesis advisor Dr. Bikramjit Banerjee, for all the help and support during this research journey. I would also like to thank my friends Pujan, Gokul and Nabin for their support.

This work was supported in part by National Science Foundation grant IIS-1526813.

Table of Contents

List of Figures

List of Abbreviations

BT	Behavior Tree
RL	Reinforcement Learning
FSM	Finite State Machine
UAV	Unmanned Aerial Vehicle
ROS	Robot Operating System
LOR	Learning Selector Node

Chapter 1: Introduction

Unmanned Aerial Vehicles (UAVs) have increasingly been used in a wide scope of applications, such as surveillance [1], farming [2], cartography [3], disaster management [4], wildfire tracking [5], cloud monitoring [6], structure supervision [7] and many more. As they can quickly cover an area by flying at high speed and altitude, they have been used extensively in many fields nowadays. The fact that UAVs can reach areas that are not feasible to people has made them the first choice to use even in sensitive applications such as first aid and emergency monitoring.

UAVs are helping in various sectors; hence, people are interested to combine multiple UAVs to solve a problem more efficiently. Instead of using a single UAV, using multiple UAVs simultaneously can be extremely efficient as they can complete a task quicker and also cover a larger area. Multiple UAVs can accomplish far more in terms of mission duration, mission area, and even mission payload balancing [10], compared to a single UAV. Besides, they would be proficient in a variety of complex tasks that need a specific degree of collaboration and cooperation. This proficiency makes them ready to achieve a bigger range of tasks, which is achieved by working alongside each other by sharing vital information needed for the effective completion of the mission. The use of a single UAV is considered as a reduction of the system usefulness; since it has many flaws in various features compared to multi-UAV systems [8]. In recent years, a lot of work has been carried out to work on cooperative control for UAVs such as formation flying [17]. Still, some cooperative control problems need to be addressed such as coordinated target assignment and intercept [9].

The capacities and jobs of multi-UAVs are advancing and require new ideas for their control [11]. New methods in arranging and execution are required to facilitate the operation of a fleet of UAVs. An overall control system architecture must likewise be built that can perform coordination of the UAVs and quickly reconfigure to account for changes in the environment of the fleet [12].

One such control system architecture used to control autonomous agents is Behavior Trees (BT). BTs were used traditionally in the computer gaming industry to solve the problem faced by complex control structures due to the ever-increasing complexity of game code. In computer science, control flow is the order in which individual statements, instructions, or function calls of a program are executed or evaluated. These programs use statements that change a program's state. Since the state space of games started increasing, it was not feasible to use pre-existing control architectures such as Finite State Machines (FSM).

There are still some shortcomings with BTs and their nodes. Even though a BT greatly improves the performance with growing complexity, it is still a fixed-rule decision model. To solve the problem of how to select actions and behaviors efficiently by autonomous agents in BTs, authors of [19] have come up with a way to incorporate Reinforcement Learning (RL) techniques in BT. This technique makes use of the RL algorithm so that an agent can learn an optimal policy which can then be used to make an efficient action selection. This existing technique for BTs, however, only works for a single agent problem. But there is no guidance on how to apply it in multi-agent problems. One way would be for each agent to ignore the other agent and carry out their respective works. However, this would fail to exploit the cooperative nature of our application.

The goal of this project is to study how multiple UAVs should apply RL in a cooperative setting to learn individual BTs. Hence, we suggest a novel solution to that problem by adding communication features to the RL state space and leveraging the Robot Operating System (ROS) message passing architecture that allows agents to take messages into account when conducting RL. We hope to understand whether the addition of extra communication helps agents to perform better in solving a problem.

Chapter 2: Background

2.1 Finite State Machines (FSMs)

An FSM is one of the basic mathematical models of computation. It consists of a set of states, transitions, and events, as shown in Fig. 1, which shows an example of an FSM designed to carry out a grab-and-throw task [14]. FSMs have been the standard choice of control structure while designing a task-switching structure for a long time [13]. FSMs are intuitive structures to understand. However, it is extremely difficult to manage once the scale of the program is increased. Historically, FSMs have also been used as a control architecture for many autonomous agents. An autonomous agent needs to be able to quickly and efficiently react to changes. Similarly, agents should enable the components that are developed to be independently tested to increase the modularity. By modular, we mean the degree to which a system's components may be separated into building blocks, and recombined [22].

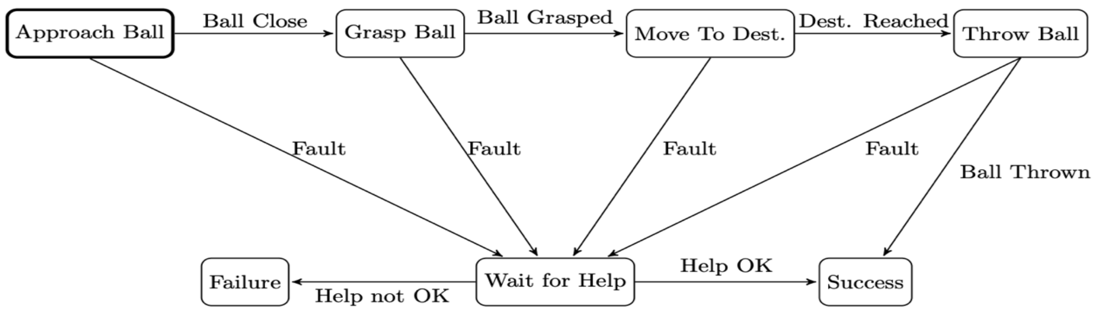


Fig. 1: Graphical representation of an FSM designed to carry out a simple grab-and-throw task. The initial state has a thicker border, and events names are given next to the corresponding transition arrows [14].

Since the complexity grows with the size, it is easier to develop the components, test it independently from each other. It has been argued in [14] that, unfortunately, there is a tradeoff between reactivity and modularity that is inherent to FSMs. The state transitions on an FSM are *one-way control transfers*. The shortcomings of *one-way control transfers* have been discussed by the author of [15]. This inherent limitation of FSM is not scalable requiring many transitions between components.

To combat the deficiency of FSM, people started using better control architecture such as BT. Isla puts forward the BT model [16], which significantly improves development efficiency. Using BTs instead of FSMs to switch tasks allows us to define actions as separate modules. Since BTs are executed in a certain way, it makes it reactive to any changes that may come in the way. Hence, BTs have become the most common architecture for autonomous agents and gaming industries.

2.2 Behavior Trees (BTs)

In simple terms, a BT is a paradigm in which we can structure the switching between different behaviors/tasks/actions in an autonomous agent. Computationally, a BT is a directed rooted tree, with internal nodes serving to direct control flow, while leaf nodes represent action execution or condition evaluation.

The root is the top of the tree with no parent, while every other node has at least one parent. There are two types of internal nodes: *control flow nodes and execution nodes. Control flow nodes* have at least one child, whereas, the *execution nodes* do not have children. There are mainly three types of *control flow nodes*: *selector node, sequence node,* and *parallel node*. There are mainly two types of *execution nodes*: *action node and condition node*. Fig. 2 shows the various types of nodes used in this paper.

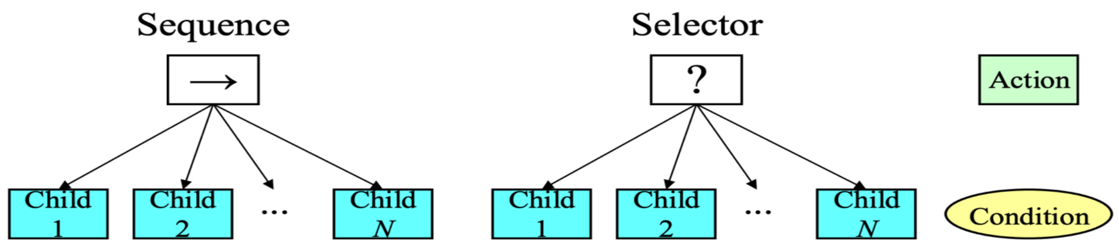


Fig 2: Various node types of a behavior tree

Behavior trees start their execution from the root node passing ticks with a given frequency and then moving onto their children from left to right. The node is executed only if the tick is reached there. After that, the child may pass the tick onto its children, but they must return to the parent one of three status: *Running* (if the execution is not finished and has to continue), or *Success* (if the goal has successfully been achieved) or *Failure*.

The *sequence node* routes the ticks to its children from left to right until one of its children returns either *Failure* or *Running.* After that, it returns *Failure or Running* accordingly to its parent. In this case, the remaining children do not receive the tick. It returns *Success* if and only if all of its children return *Success*.

The *selector node* routes the ticks to its children from left to right until one of its children returns either *Success* or *Running.* After that, it returns *Success* or *Running* accordingly to its parent. In this case, the remaining children do not receive the tick. It returns *Failure* if and only if all of its children return *Failure.*

The *parallel node* route the ticks to its children and it returns *Success* if M children return *Success.* It returns *Failure* if N - M + 1 children return *Failure.* It returns *Running* otherwise. Here, N is the number of children and M ≤ N is a user-defined threshold.

Similarly, an *action node*, when it receives a tick, executes a predefined command and it returns *Success* if/when the (multi-step) action is correctly completed. It returns *Failure* if the action fails to complete and it returns *Running* if it needs more steps to complete the task. When a *condition node* receives a tick, it examines a certain premise and returns *Success* or *Failure* based on its evaluation. The *condition node* never returns *Running.*

There is an additional type of *control flow node,* called the *Decorator.* It has a single child and it alters the return status of its child according to a certain user-defined logic and also selectively ticks the child according to some predefined rule [18].

To help and avoid the unwanted re-execution of some nodes, nodes with memory have also been introduced. The main difference between the normal *control flow nodes* and the *control flow nodes* with memory is that the latter one can remember or store whether a child has returned *Success* or *Failure.* This helps to avoid the re-execution of the child until the whole *sequence* or *selector node* finishes in either *Success* or *Failure*. The memory is cleared out when the parent node returns either *Success* or *Failure* so that at the next activation all children are considered. Nodes with memory can be regarded as *syntactic sugar* as the same behavior can be achieved by making use of a combination of normal *control flow nodes.*

2.3 Reinforcement Learning (RL)

RL is a paradigm that allows an agent to learn from trial and error with no prior understanding of the environment. In simple terms, it is to learn how to map situations/states to optimal actions so that it can perform better and maximize a numerical reward signal. In this type of learning, the learner is not told what to do and what actions to take but instead must figure out which action to perform so that they can accumulate the most reward by taking those actions.

Technically, RL problems are categorized as *Markov Decision Processes* or MDPs. MDPs are a classical formalization of sequential decision making, where actions impact not just immediate rewards, but also subsequent states or situations, hence, ultimately affecting future rewards. In this sense, MDPs involve delayed reward and the need to tradeoff immediate and delayed rewards [20].

Here, the learner or the one to make the decision is called an *agent*. The *agent* interacts with the outside world and everything outside the *agent* is known as the *environment.* So, the *agent* continuously interacts with the *environment* by selecting certain actions to perform and then the *environment* responds to these actions and presents the *agent* with a new situation or state to face. Moreover, the *environment* also provides an agent with special numerical values known as the *reward.* The agent then seeks to maximize that *reward* over time by its choice of actions. Fig. 3 shows how the agent and environment interact in MDPs.

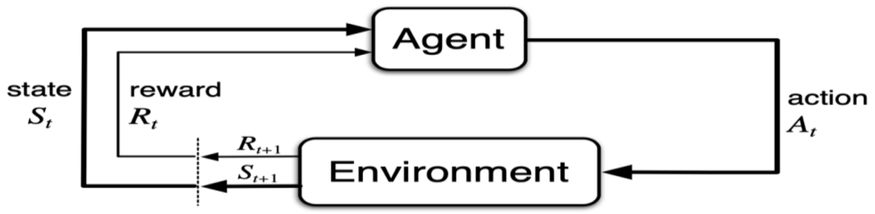


Fig. 3: The agent–environment interaction in a Markov decision process [20]

An MDP is given by the tuple < *S*, *A, R, P* >, where *S* denotes the set of visible environmental states that an *agent* can be in at any given time *t*. *A* is the set of actions that the *agent* can choose from at any state. In each discrete time step *t,* the *agent* senses the current state s_t selects a particular action *a* and executes it. Then, the *environment* responds to the action performed by the *agent*, returns the reward *R* and also generates a successor state s_{t+1}. $R: S \text{ X } A \rightarrow R$ is the reward function, that is., $R\ (s,\ a)$ specifies the reward from the environment that the *agent* gets for executing action $a \in A$ in state $s \in S$. $P: S\ X A\ X S \rightarrow [0,\ 1]$ is the state transition probability function which specifies the probability of reaching the *next state* in the Markov chain with that particular *action* selected by the *agent*.

The main goal of the *agent* is to learn a policy $\pi : S \rightarrow A$ that maximizes not the immediate reward, but the cumulative reward in the long run. This is based on the currently observed state s_t to select the next action a_t. To be able to maximize the sum of current and future rewards, almost all RL algorithms involve estimating *value functions* defined by V^{π}.

The *value function* of a state *s* under a policy π is denoted by $\upsilon_{\pi}(s)$ and defined as the expected sum of the rewards when starting in *s* and following π thereafter. Formally we can define $\upsilon_{\pi}(s_t) = E_P\,[R(s_t,\ \pi(s_t)) + \gamma\ R(s_{t+1},\ \pi(s_{t+1})) + \ldots]$ where $s_t,\ s_{t+1},\ \ldots$ are successive samplings from the distribution *P* following the Markov Chain with policy π [3]. Here, γ is a parameter, $0 \leq \gamma \leq 1$, called the discount rate which determines the present value of future rewards. Similarly, *action-value* function for policy π given by *Q* is defined as:

$$Q(s,\ a) = R(s,\ a) + \gamma\ \max_{\pi} \textstyle\sum_{s'} P(s, a, s')V\pi(s') \qquad (1)$$

We define *Q*(*s, a*) *as* the value of taking action *a* in state *s* under a policy π, which is the expected sum of the rewards when the agent starts from state *s* at step *t*, executes the action *a*, and thereafter follows the optimal policy. The final optimal policy is calculated as

$$\pi(s) = \arg\max_{a} Q(s, a) \tag{2}$$

This means that we want the *agent* to learn a policy π maximizing *Q(s, a)* for any state *s*.

For our experiments, we have focused on tasks that allow *Q(s, a)* values to be stored in a table, i.e, tabular Q-learning. For those tasks, the state space is designed simply so as not to require any function approximation [18]. The algorithm to update the *Q(s, a)* is provided below.

<u>Q-Learning Algorithm</u>

```
Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode)
    Initialize $s$
    Repeat (for each step of episode)
        Choose $a$ from $s$ using policy  derived from $Q$
        Take action $a$, then observer $r, s'$
        $Q(s, a) \leftarrow Q(s, a) + [\ R(s, a) + \max_{a} Q(s', a) - Q(s, a)\ ]$
        $s \leftarrow s'$
    until $s$ is terminal
```

Here, α is the learning rate which is set between 0 and 1. Lower the value of α, the slower the *agent* learns. Hence, if we set a higher value such as 0.9, the agent would learn quickly.

2.4 Learning Selector Node

For our experiment, we have used a special kind of node called a *learning selector node* (LOR) designed in [19]. In the definition of the behavior trees, *selector nodes* will choose their children from left to right until it finds a certain node that returns either *Success* or *Running.* This formulation puts more weight on the left-most (first) child as it will always be ticked first and that should not always be the case. Sometimes the order of the children nodes may not be known before execution. Consider, for instance, the task of a UAV attempting to find a target. In this case, it may not be known which of many alternative ways to find a target should be attempted first. This choice may even vary from location to location. For instance, in the vicinity of tall buildings, it may be beneficial to elevate first to get a better view of the interspaces. Whereas in open areas, it may be useful to rotate first to quickly locate a target. Fu et al. [19] suggest that we attach the weight to the child nodes and then the *selector node* be modified in such a way that it ticks child nodes in accordance with the weights from high to low. The authors suggested a way to use the idea of RL to optimize selectors naming them *learning-selector*. In our paper, we have used the following Learning Selector algorithm to use the *learning selector node* as a part of our BT.

<u>Learning Selector Algorithm</u>

```
LearningSelector() function return status
    observe current state s
    for each action a and current s sort Q(s,a)
    disturb this order with a probability e
        for each child node[i]
        childStatus <- Tick(node[i])
        if childStatus equals RUNNING
            return RUNNING
        elseif childStatus equals SUCCESS
            receive the current reward r
            observe current state s'
            update Q value
            return SUCCESS
        end
    end
    return FAILURE
end
```

This way, when a *learning-selector node* is ticked, it will evaluate the current state of the agent. After that, it will sort the state-action pair with corresponding *Q(s, a)* within a certain state. It then chooses to tick a random child with the probability of epsilion *e* and then tick the child with the highest value of *Q(s, a)* with the probability of 1-*e* and update the *Q* table.

2.5 Robot Operating System (ROS)

ROS is a framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms. Since it is an open-source project, we can use the packages developed by other people. One of the main building blocks of the ROS is called a node. Nodes are the executable files within a ROS package. Nodes have the special property to communicate with other nodes using *Topics and Services. Topics* are a way for nodes to send messages to each other. In our paper, we have made use of message passing between two nodes to demonstrate the communication between two UAVs. Fig. 4 shows how message passing using Topics works via publication and subscription. Messages are routed via a transport system with publish/subscribe semantics. A node sends out a message by publishing it to a given topic. A node that is interested in a certain kind of data will subscribe to the appropriate topic [21]. There can be multiple concurrent publishers and subscribers for a single topic.

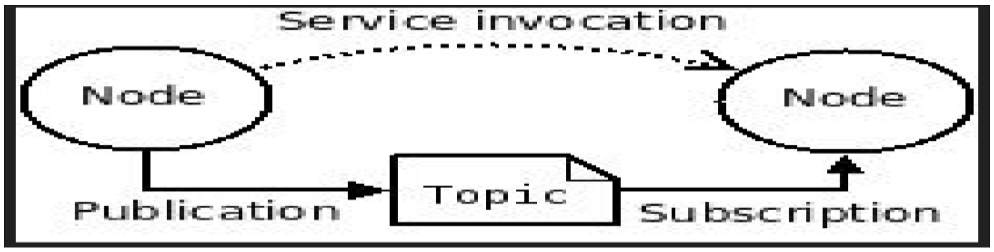
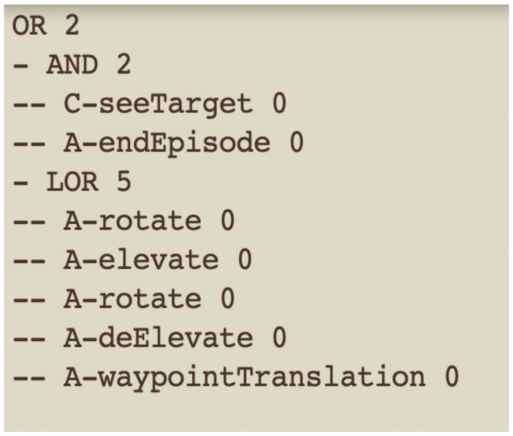


Fig. 4: Message Passing between two nodes using *Topics*

Chapter 3: Methodology

3.1 BT Framework

For our project, we have built a BT framework that serves as the main unit needed to control the UAV. This framework adheres to the definition and functionality of the BT defined in section 2.2. It has two main functions, namely, a BT building function that builds the behavior tree off a text file as shown in Fig. 5 and an execution function that executes the behavior tree. It can also call actions designed for a particular agent.



```
OR 2
- AND 2
-- C-seeTarget 0
-- A-endEpisode 0
- LOR 5
-- A-rotate 0
-- A-elevate 0
-- A-rotate 0
-- A-deElevate 0
-- A-waypointTranslation 0
```

Fig. 5: A text file with the representation of BT

The program starts by reading a text file that contains information to build a BT. After the BT is built, the program starts executing the BT. The leaf node as discussed before is either a *condition node* or an *action node*; hence, at the end of the execution, the BT framework calls the behaviors defined for the particular agent. In our case, we have 5 actions and 1 condition in our BT which will be discussed in the section 3.3.

The Fig. 6 represents a tree structure that will be built using a text file shown in Fig. 5. The number of dashes in Fig. 5 represents the depth of the node in the tree. The root node has no dash associated with it. The root node is a *selector node* (*OR*) which has 2 children. Its first child (the second line having one dash) is a *sequence node* (*AND*) which has 2 children of its own. Its first child is a *condition node (C-seeTarget)* and its second child is an *action node (A-endEpisode)*. Since they are execution nodes, they do not have children. We then reach the second child of the root node, which is a *learning selector node* (*LOR*), which has 5 children of its own. In this way, the entire tree is built.

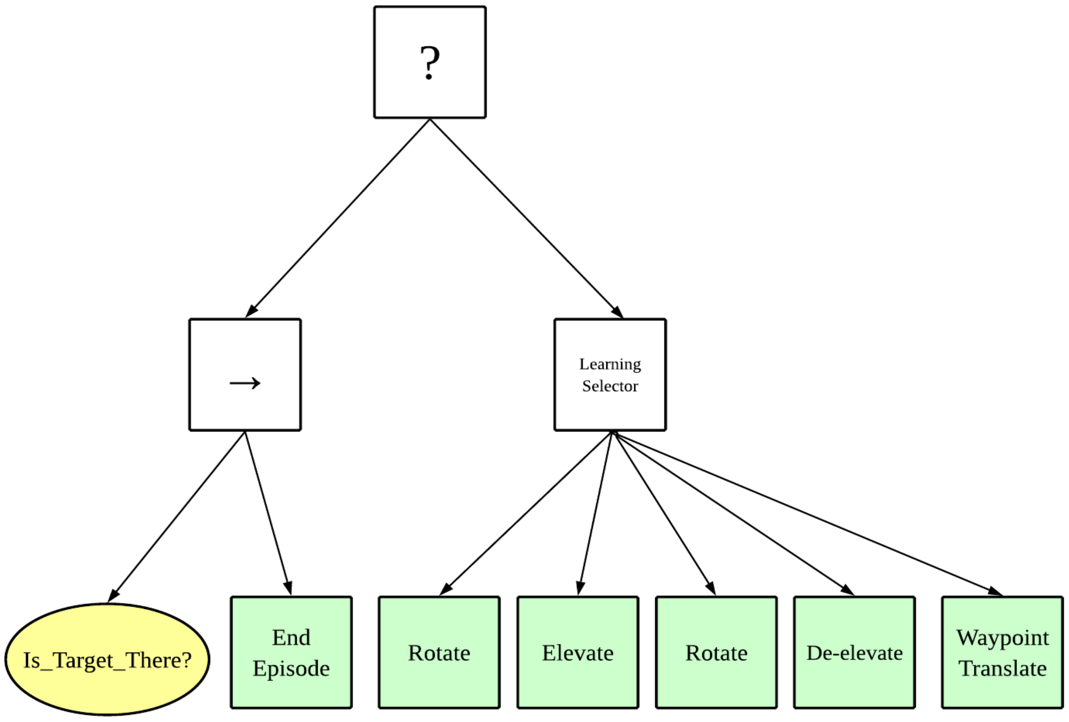


Fig. 6: A graphical representation of a BT used in our experiment corresponding to the text specification in Fig. 5.

3.2 Pseudo Environment

For our project, we have carried out experiments in pseudo environment which closely resembles the actual environment. The pseudo environment consists of a 10 by 10 grid which is acquired by defining a 2-dimensional array of size 10 by 10. The environment represents the area where the UAVs will fly and try to find the target. It also comprises occlusions, mirroring real-world settings, which will affect the vision of the UAVs.

The list of waypoints is associated with its corresponding coordinates in the grid. The waypoints will be necessary for UAVs to move around. One way to choose the set of waypoints is by following the deployment algorithm proposed in [23] which gives a set of points in a space such that the entire area is visible. Since the environment we designed is fairly small, we have developed a lookup table for UAVs. This lookup table returns a set of visible blocks, considering the occluded grids, from any location within the environment. In our experiment, each possible combination of direction, height, and waypoint is regarded as a distinct state.

For example, once the UAV reaches a certain state, a function will be called which will return a set of visible coordinates from that state. In this way, UAVs will be able to figure out if they can see the target or not. For our experiment, we have developed the grid world environment of three different variations. First, we considered the case of having one fixed target location where a single UAV is flying. Fig. 7 shows this case.

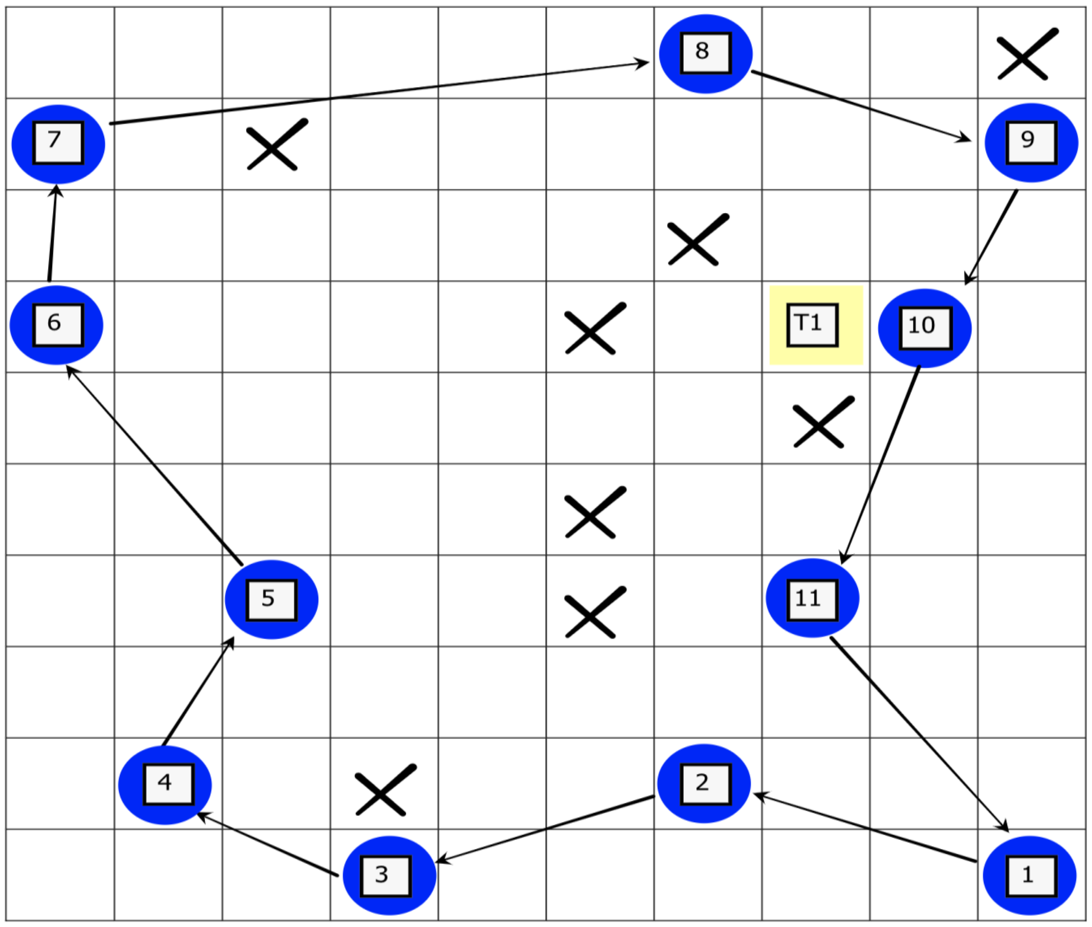


Fig 7: 10 X 10 grid-world environment consisting of waypoints and one target location for a single UAV

Here, a grid cell, with a yellow square labelled T1, is a target location. Similarly, grid cells, labelled X, are the occlusions present in the environment. Grid cells, with a blue circle labelled with numbers, are the waypoints. As shown in the figure, the UAV starts from waypoint one and follows the numbers in ascending order. After it reaches the end, it again starts from the beginning. The target T1 is visible from state of tuple < waypoint #10, low-height,orientation-WEST>.

Similarly, for the second experiment, we consider the case of having two target locations where a single UAV is flying. Here, we incorporate the stochasticity in the environment, as there is an equal probability for the target to be in either of the two potential locations. Fig. 8 shows this case.

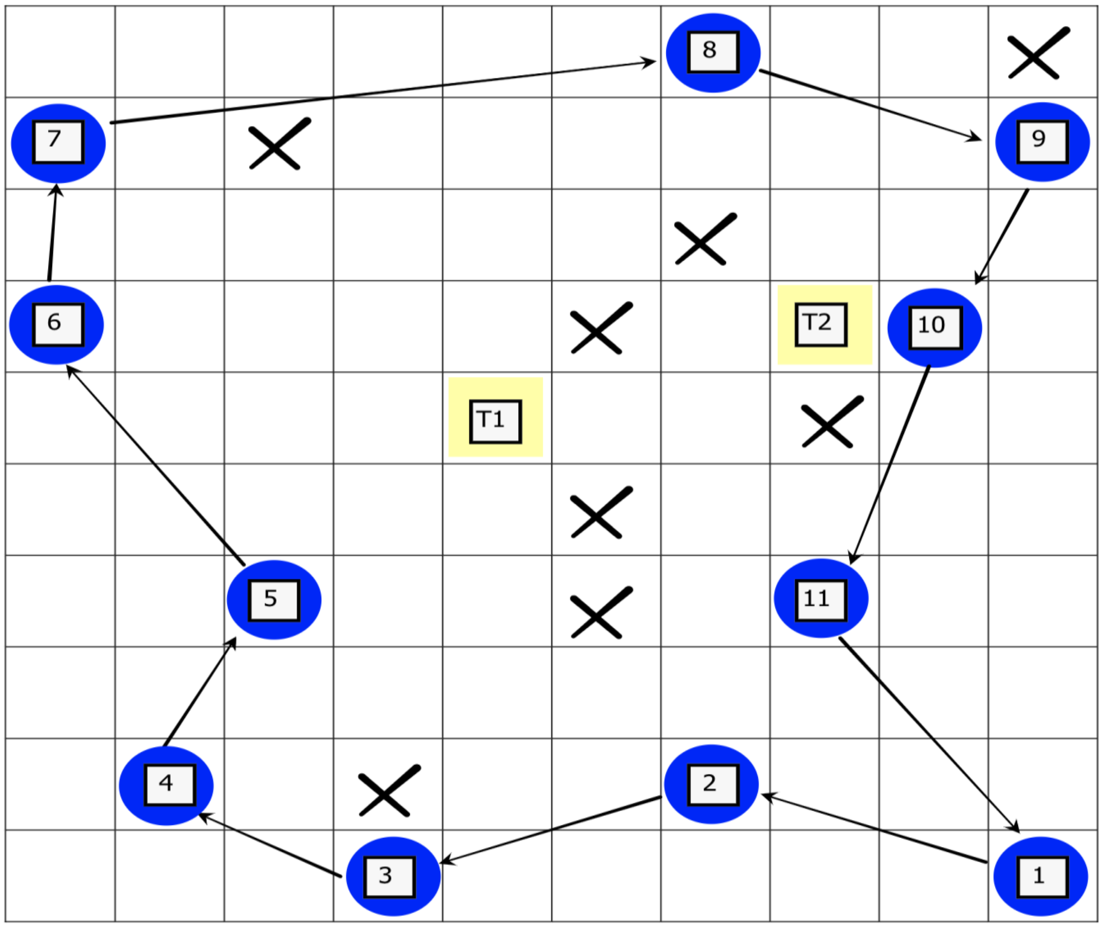


Fig. 8: 10 X 10 grid-world environment consisting of waypoints and two targets locations for one UAV

Here, a grid cell, with a yellow square labelled T1, is the first target location. Similarly, a grid cell, with a yellow square labelled T2, is the second target location. The two target locations have an equal probability to be chosen as the target location for a particular run. Grid cells, labelled X, are the occlusions present in the environment. Grid cells, with a blue square labelled with numbers, are the waypoints. As shown in the Fig. 8, the UAV starts from waypoint number one and then follows the numbering in ascending order. After it reaches the end, it again starts from the beginning. The target T1 is visible from state of tuple < waypoint #6, high-height, orientation-EAST >. The target T2 is visible from state of tuple < waypoint #10, low-height, orientation-WEST >.

Lastly, we consider the case of having two UAVs and two target locations. There is an equal probability for the target to be in either of the two potential locations. This is the case where coordination between two UAVs takes place. Fig. 9 shows this case.

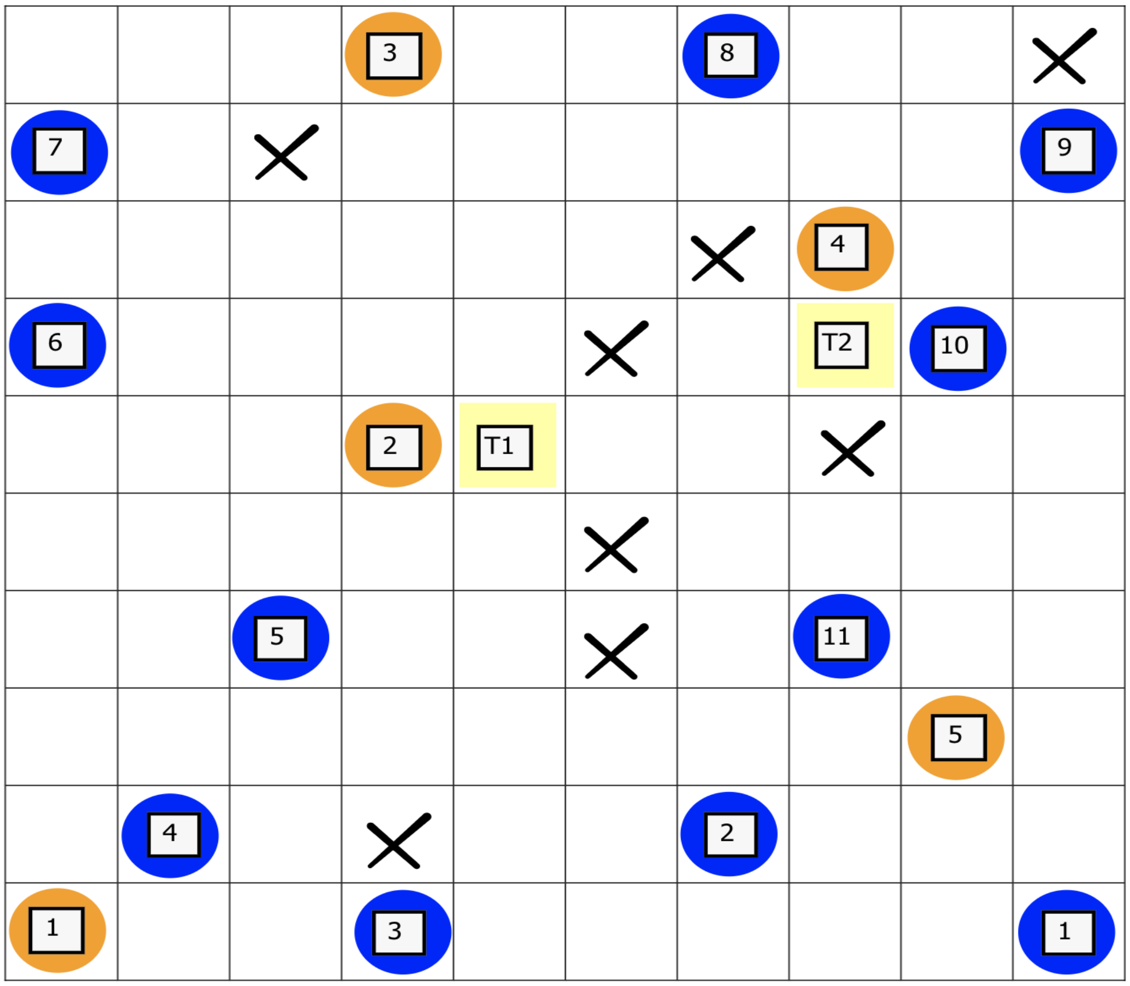


Fig. 9: 10 X 10 grid-world environment consisting of waypoints and two targets locations for two UAVs

Here, a grid cell, with a yellow square labelled T1, is the first target location. Similarly, a grid cell, with a yellow square labelled T1, is the second target location. The target T1 is visible from state of tuple < waypoint #6 (with blue square), high-height, orientation-EAST > and tuple < waypoint #2 (with orange square), low-height, orientation-EAST >. The target T2 is visible from state of tuple < waypoint #10 (with blue square), low-height, orientation-WEST > and tuple < waypoint #4 (with orange square), low-height, orientation-SOUTH >.

Grid cells, labelled X, are the occlusions present in the environment. Grid cells, with a blue square labelled with numbers, are the waypoints for the first UAV. The UAV starts from waypoint number one and then follows the numbering in ascending order until it reaches the last one, i.e., 11. After reaching the end, it again starts from the beginning. Similarly, grid cells, with an orange square labelled with numbers, are the waypoints for the second UAV. The UAV starts from waypoint number one and then follows the numbering in ascending order until it reaches the last one, i.e., five. After reaching the end, it again starts from the beginning.

3.3 Behaviors/Actions

We have come up with several actions that the UAV carries out for the effective search of the target. For our case, we have defined 4 actions, namely, Rotate, Elevate, De-elevate, and Waypoint Translation. The details of the four actions are as follows:

a. Rotate

This action makes the UAV rotate by 90 degrees each time it gets ticked. For example, if the UAV is facing the north direction, then one tick turns it towards the east direction. It always follows a predefined pattern, turning from east to south, south to west,

west to north and north to east. This action is the only action that returns *Running* status in our experiment. It returns *Running* until the UAV returns to direction it was facing when it started. After that, it returns *Success*. In this manner, the UAV completes a 360 degrees rotation.

b. Elevate

This action, each time it gets ticked, makes the UAV climb up if it is in a *lower* height. For example, if the UAV is in a *low* height, it takes the UAV to an *elevated* height and returns *Success*. But if the UAV is already at an *elevated* height, then it continues to remain in the same height and returns *Success.* It does not return *Running* and *Failure*.

c. De-elevate

This action, each time it gets ticked, makes the UAV climb down if it is in an *elevated* height. For example, if the UAV is in an *elevated* height, it takes the UAV to a *lower* height and returns *Success*. But if the UAV is already at a *lower* height, then it continues to remain in the same height and returns *Success.* It does not return *Running* and *Failure*.

d. Waypoint Translation

This action, each time it gets ticked, makes the UAV go to the next waypoint. Waypoints are coordinates in our grid-world example as discussed in section 3.2. At the start of the program, we have defined certain locations to be the waypoints in a systematic order. So, the waypoint translation makes UAV move in that order until it reaches the last waypoint. After the UAV reaches the last waypoint, it again starts from the first waypoint.

Chapter 4: Experimental Evaluation

In our project, we have carried out experiments on three different levels to measure the advantages of adding extra communication features in the RL state-space to support the cooperative nature of our application.

4.1 One UAV and one target location without communication

First, we experimented to see how a single agent learns its BT with just waypoints, direction, and height as an RL state. There was only one target location present in the environment. This experiment was done adhering to the concepts discussed in [19], where we used the existing RL technique for BTs which worked for single-agent problems. The motivation behind this experiment was to validate the RL in *learning selector node* (LOR) as a preliminary step, and ensure that it worked as expected, before using it in other experiments.

4.2 One UAV and two target locations without communication

Second, we experimented to see how a single agent learns its BT with just waypoints, direction, and height as an RL state. For this experiment, we added a second target location in our environment and made sure that at each iteration either of the two target locations was chosen to be a goal location. The purpose of this experiment was to establish a baseline for the performance of a single UAV in locating targets when it ignores the presence of a second UAV. In this setting, it is unable to exploit any cooperative communication with the second UAV that may impact its task.

4.3 Two UAVs and two target locations with communication

We carried out our novel experiment to see how the addition of extra communication features to the RL state space can affect the way the agent learns its BT.

For this experiment, agents learned their BTs with waypoints, direction, height, and communication feature as an RL state.

We included a second UAV having its own set of waypoints in the same environment setup. We call this UAV2 and we call the first UAV1. The program was compiled and two ROS nodes were obtained. One node represented the UAV1 discussed in sections 4.2 and 4.3. The next node represented the newly incorporated UAV2. These two nodes were executed in two separate terminal windows.

In our experiment, we made UAV2 go through fewer waypoints so that it could see targets quicker than UAV1. The extra communication feature was maintained as a vector of size equal to the number of the opposing UAV's waypoints. For example, if UAV1 went through 11 waypoints and UAV2 went through five waypoints, then the size of the extra communication feature vector of UAV1 would be five and that of UAV2 would be 11.

During the execution, as UAV2 saw the target from a certain waypoint, it passed a message to UAV1. The message included a flag that contained a value of one if the target was found or zero if the target was not seen from the waypoint. It also sent the index number of the waypoint where it was so that the extra feature vector of UAV1 could be updated as necessary. In this way, communication was achieved.

As seen in Fig. 9, if the target was not at the first location T1, UAV2 would send a message (with the flag of zero and index of one) to UAV1, since UAV2 could only see the first target from its second waypoint (orange circle labelled two). After UAV1 received this message, it updated its communication feature vector with the appropriate flag.

Chapter 5: Results

The experiments were conducted to observe and compare the difference in how the agent learns its BT with and without communication. We plotted the learning curves as number of iterations vs. length of episodes. First, we have shown the result from section 4.1. Then, we have shown the combined result of sections 4.2 and 4.3, where we compared and analyzed the graphs obtained from learning with and without communication.

5.1 One UAV and one target location without communication

We plotted a learning curve to show that our implementation of the technique mentioned in [19] is correct. We can see in Fig. 10 that our agent is learning its BT with time since episode length is decreasing as number of iterations increases. At the start of the program, the episode length was 42 but with time and learning, the agent has learned its BT well and decreased episode length to 16. This verifies that our implementation of the RL technique for BTs proposed by [19] is correct.

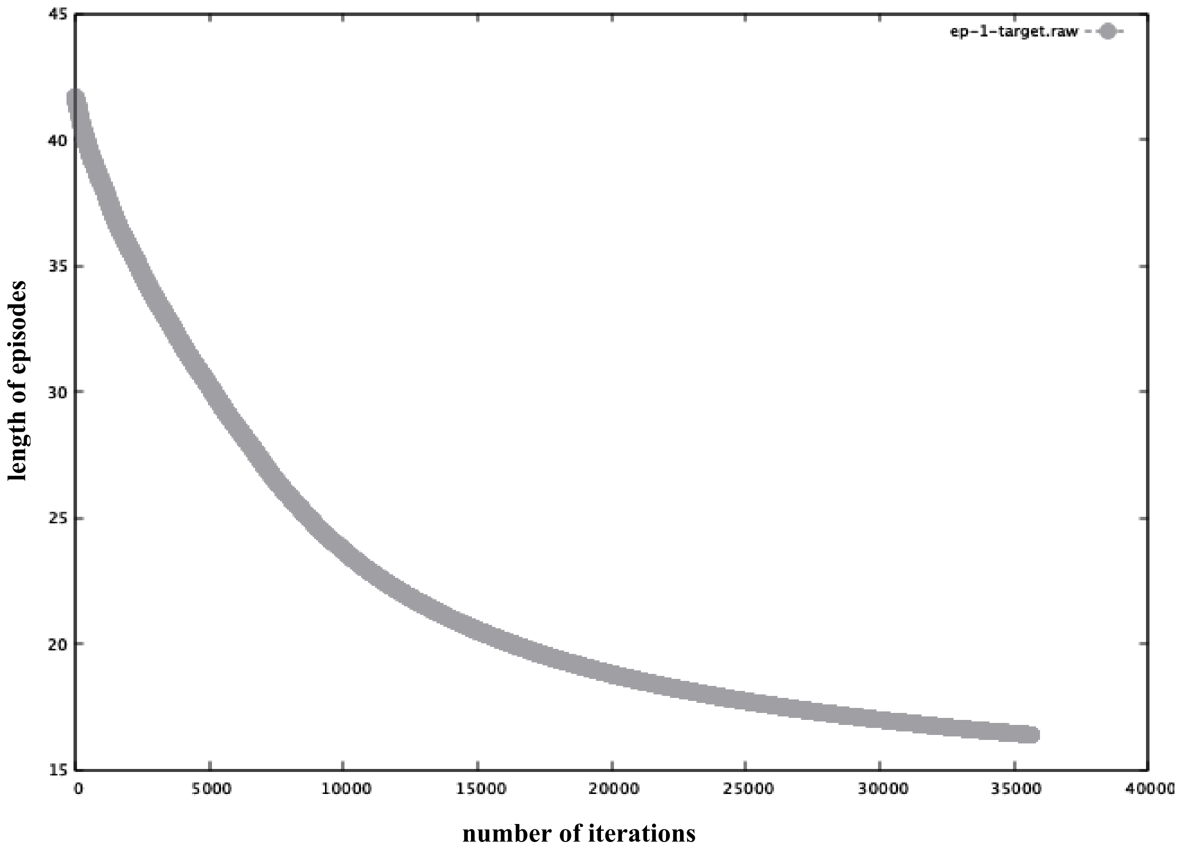


Fig. 10: The learning curve of UAV involving single target and no communication averaged over 4 independent trials.

5.2 Comparison between learning without communication and with communication

In this section, we compared and analyzed the graphs of the agents learning with and without communication. Both curves in Fig. 11 were obtained as averages of the data from four independent trials.

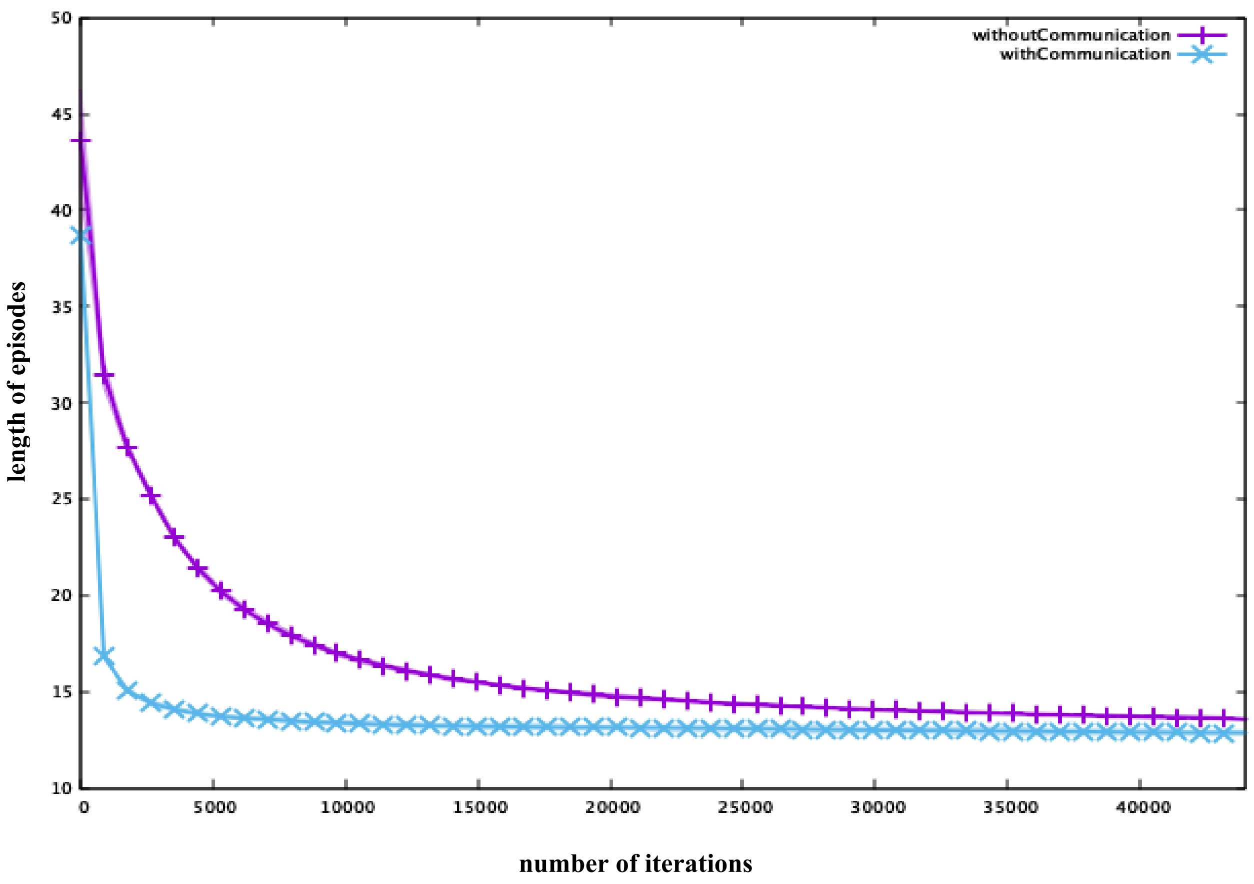


Fig. 11: The learning curve comparison between learning with communication and without communication averaged over 4 independent trials. This scenario involved two potential locations for the target, representing a searcher's uncertainty about where the target might actually be located.

From the graph, we see that a learning curve with communication (blue) is lower than the learning curve without communication (purple). When the program started, the average episode length of learning without communication was 43 and the average episode length of learning with communication was 38.

In the absence of the target in the first potential location, the agent learned to use the message, quickly bypassed the initial target location and proceeded to the next potential location, which caused the episode length to significantly drop to 17. However, the episode length in learning without communication slowly decreased as the number of iterations increased.

We see that the agent benefited from learning with communication for the first 30,000 iterations. As observed in Fig. 11, the episode length was similar for learning with and without communication after the the first 30,000 iterations. We see that the episode length for both curves converges to 15 towards the end of the experiment.

Chapter 6: Discussion

The analysis of the graphs obtained from learning with and without communication reveals new avenues for UAVs to use RL to learn their BTs effectively. We discovered that the addition of extra communication features in the RL state space could help UAVs learn better policies. We demonstrated that we can leverage the ROS message passing feature to establish a cooperative environment for multiple UAVs. In this way, they can collaborate to solve a problem in less time. We expected that adding the extra communication feature would help UAV to change its choice of action, if the target could not be found at the first potential location. After receiving the message from UAV2, we expected that UAV1 would learn its BT differently and attempt to reach the second target location faster, which would ultimately decrease the episode length.

Until the first 30,000 iterations in the experiment, the episode length was significantly lower for learning with communication as compared to that of learning without communication. The lower episode length signifies that the agent found the target quickly after communication was established. The decrease in the episode length indicates that the agent learned to use the message whenever the target was not present in the first potential location, quickly bypassed the initial location, and proceeded to the next possible location.

Even in the absence of communication, the agent was able to learn its BT, however, adding communication feature helped the agent learn its BT in less time. This observation shows that the communication feature makes UAVs better adapt to changes ultimately affecting how they learn their BTs. In addition to the communication between the UAVs, the reactiveness of BT has further boosted the cooperative nature of our application. Even in a short period of decision time, UAVs can quickly switch between actions if they get information about changes in the environment.

In the experiment, we observed that learning with communication was not effective after the first 30,000 iterations. The ROS nodes of both UAV1 and UAV2 were executed for the same number of ticks. However, since the number of waypoints was less for UAV2 than for UAV1, the UAV2 ROS node might have finished its execution ahead of the UAV1 ROS node. As a result, the communication stopped, which might have caused the curves to converge towards the end of the experiment.

Chapter 7: Conclusion and Future Work

This research studied how multiple UAVs can effectively apply RL in a cooperative setting to better learn their BTs. The communication was established with the help of the ROS message passing feature to create a cooperative nature between two agents. Our findings suggest that cooperation between agents can help them learn effective RL policy. The RL policy ultimately helps agents learn their BTs. This observation was made by comparing the learning curves of two types of learning done by an agent, namely, learning with communication and learning without communication. The comparison demonstrated that episode length was lower in learning with communication, which suggests that the inclusion of extra communication features in RL state-space assist UAVs to better learn their BTs.

As we conducted our experiment in a pseudo-environment, we plan to extend this work further by testing our solution in a Gazebo simulator which closely resembles the real-world settings. This will provide a more rigorous testing framework for our solution. We also believe that it is extremely important to explore whether the learning pattern would still be effective after increasing the number of agents.

References

1. Basilico, N.; Carpin, S. Deploying Teams of Heterogeneous UAVs in CooperativeTwo-level Surveillance Missions. In Proceedings of the 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Hamburg, Germany, 28 September–2 October 2015; pp. 610–615.
2. Lottes, P.; Khanna, R.; Pfeifer, J.; Siegwart, R.; Stachniss, C. UAV-Based Crop and Weed Classification for Smart Farming. In Proceedings of the 2017 IEEE International Conference on Robotics and Automation (ICRA), Singapore, 29 May–3 June 2017; pp. 3024–3031.
3. Cesetti, A.; Frontoni, E.; Mancini, A.; Ascani, A.; Zingaretti, P.; Longhi, S. A visual global positioning system for unmanned aerial vehicles used in photogrammetric applications. *J.Intell.Robot.Syst.***2011**,*61*,157–168.
4. Maza, I.; Caballero, F.; Capitán, J.; Martínez-de Dios, J.R.; Ollero, A. Experimental results in multi-UAV coordination for disaster management and civil security applications. *J.Intell.Robot.Syst.***2011**,*61*,563–585
5. Pham, H.X.; La, H.M.; Feil-Seifer, D.; Deans, M. A Distributed Control Framework for A Team of Unmanned Aerial. In Proceedings of the 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Vancouver, BC, Canada, 24–28 September 2017; pp. 6648–6653.
6. Renzaglia, A.; Reymann, C.; Lacroix, S. Monitoring the Evolution of Clouds with UAVs. In Proceedings of the 2016 IEEE International Conference on Robotics and Automation (ICRA), Stockholm, Sweden, 16–21 May 2016; pp. 278–283.

7. Guerrero, J.A.; Bestaoui, Y. UAV path planning for structure inspection in windy environments. *J. Intell. Robot. Syst.* **2013**, *69*, 297–311.

8. Aljehani, M., Inoue, M.: Multi-UAV tracking and scanning systems in M2M communication for disaster response. In: 2016 IEEE 5th Global Conference Consumer Electronics, pp. 1–2 (2016)

9. Beard RW, McLain T W, Goodrich M (2002) Coordinated target assignment and intercept for unmanned air vehicles. In: Proceedings of the IEEE International Conference on Robotics and Automation, pp. 2581–2586. Washington

10. M.-A. Messous, S.-M. Senouci, and H. Sedjelmaci, "Network connectivity and area coverage for UAV fleet mobility model with energy constraint," *2016 IEEE Wireless Communications and Networking Conference*, 2016.

11. P. R. Chandler, S. Rasmussen, "UAV Cooperative Path-Planning," proc. of the AIAA GNC, Aug. 14– 17, Paper No. AIAA-2000-4370, 2000.

12. A. Richards, J. Bellingham, M. Tillerson, and J. How, "Coordination and Control of Multiple UAVs," *AIAA Guidance, Navigation, and Control Conference and Exhibit*, May 2002.

13. Matthew Powers, Dave Wooden, Magnus Egerstedt, Henrik Christensen, and Tucker Balch. The Sting Racing Team's Entry to the Urban Challenge. In Experience from the DARPA Urban Challenge, pages 43–65. Springer, 2012

14. M. Colledanchise, "Behavior Trees in Robotics and Al," 2018.

15. E. W. Dijkstra, “Letters to the editor: go to statement considered harmful,” *Communications of the ACM*, vol. 11, no. 3, pp. 147–148, Jan. 1968.

16. Isla, Damian. "Handling complexity in the Halo 2 AI."*Game Developers Conference”*. Vol. 12. 2005.

17. Giulietti F, Pollini L, Innocenti M (2000) Autonomous formation flight. IEEE Control Systems Magazine 20:34–44

18. B. Banerjee, “Autonomous Acquisition of Behavior Trees for Robot Control,” *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2018.

19. Y. Fu, L. Qin, and Q. Yin, “A Reinforcement Learning Behavior Tree Framework for Game AI,” *Proceedings of the 2016 International Conference on Economics, Social Science, Arts, Education and Management Engineering*, 2016.

20. R. S. Sutton, “Introduction: The Challenge of Reinforcement Learning,” *Reinforcement Learning*, pp. 1–60, 2017.

21. Robot Operating System Documents. [Online]. Available: http://wiki.ros.org/ROS/Concepts. [Accessed 02-April-2020].

22. JK Gershenson, GJ Prasad, and Y Zhang. Product modularity: definitions and benefits. Journal of Engineering design, 14(3):295–313, 2003.

23. A. Savkin and H. Huang, “Proactive Deployment of Aerial Drones for Coverage over Very Uneven Terrains: A Version of the 3D Art Gallery Problem,” *Sensors*, vol. 19, no. 6, p. 1438, 2019.