Combining Learning With Knowledge-rich Planning Allows For Efficient Multi-agent Solutions To The Problem Of Perpetual Sparse Rewards

by

Swastik Nandan

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved May 2022 by the
Graduate Supervisory Committee:

Theodore Pavlic, Co-Chair
Jnaneshwar Das, Co-Chair
Spring Berman

ARIZONA STATE UNIVERSITY

August 2022

ABSTRACT

This thesis has improved the quality of the solution to the sparse rewards problem by combining reinforcement learning (RL) with knowledge-rich planning. Classical methods for coping with sparse rewards during reinforcement learning modify the reward landscape so as to better guide the learner. In contrast, this work combines RL with a planner in order to utilize other information about the environment. As the scope for representing environmental information is limited in RL, this work has conflated a model-free learning algorithm – temporal difference (TD) learning – with a Hierarchical Task Network (HTN) planner to accommodate rich environmental information in the algorithm. In the perpetual sparse rewards problem, rewards reemerge after being collected within a fixed interval of time, culminating in a lack of a well-defined goal state as an exit condition to the problem. Incorporating planning in the learning algorithm not only improves the quality of the solution, but the algorithm also avoids the ambiguity of incorporating a goal of maximizing profit while using only a planning algorithm to solve this problem. Upon occasionally using the HTN planner, this algorithm provides the necessary tweak toward the optimal solution. In this work, I have demonstrated an on-policy algorithm that has improved the quality of the solution over vanilla reinforcement learning. The objective of this work has been to observe the capacity of the synthesized algorithm in finding optimal policies to maximize rewards, awareness of the environment, and the awareness of the presence of other agents in the vicinity.

ACKNOWLEDGMENTS

I am grateful to my committee members: Dr. Theodore Pavlic, Dr. Jnaneshwar Das and Dr. Spring Berman for guiding me through the process of conceptualization and formalization of the thesis statement with their knowledge. I am thankful for the committee's time, constant guidance and feedback for helping me work on a research topic that has impactful conclusions in the field of Artificial Intelligence.

I am thankful to the members of the DREAMS lab for their availability and support during brainstorming and technical discussions.

Finally, I want to thank my parents for their constant financial and emotional support without which I will have never been able to be an international student in the USA. I am thankful to my parents for their patience and dedication in taking care of my sibling, Rubai. Rubai was born congenitally deformed and has a handicap in listening and speaking. Taking care of Rubai is physically and emotionally dissipating. I am responsible for sharing the efforts of taking care of Rubai. I am forever indebted to my parents for allowing me to be away from my family to make advances in my career.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

PREFACE

In a perpetual sparse reward problem, there is no goal state that can act as an exit state for the problem. Hence, it is challenging to define a planner with only a goal state, achieving which marks the completion of the planning task and, a list of operators to change the predicates of the planning domain to achieve the goal state. In a learning algorithm, there is not much scope in describing rich environmental information to make decisions.

According to Freitas *et al.* (2014) rich environmental information such as ontology in OWL (Web Ontology Language) can be converted to planners such as Hierarchical Task Network (HTN) with is a rich representation of the environment. In this research, I have tried to qualitatively and quantitatively explore how the process of learning can benefit in its efficiency with a process of information rich planning synthesized within it. With additional information, the learning process is expected to have tweaks in finding the optimal policy faster by avoiding states that does not contribute to the overall reward. The learning process still allows for a good probability of still exploring new reward yielding states that has not been considered by the information rich planner.

INTRODUCTION

Classical reinforcement learning is designed with the objective of learning optimal policies for particular states. The goal of reinforcement-learning-like algorithms is to maximize the reward value at every time-step. There is little scope in describing rich information about the environment in their design. Deployment of reinforcement-learning-based algorithms in real-world scenarios calls for the need to improve classical reinforcement with strategies to expedite the speed of the learning process. The ability to add more information about the environment during reinforcement learning will aid in utilizing prior knowledge of the environment in searching for the optimal actions faster.

In this work, the problem of sparse reward has been explored with an attempt to improve the quality of solving sparse reward problem with classical reinforcement learning. Sparse reward problem is a category of problem which covers the domain of multi-robot coverage and multi-robot assignment. Improving the solution strategy and the quality of solution of sparse reward problem will improve the solution strategy of multi-robot coverage and multi-robot assignment problem.

From the perspective of knowledge representation, there are certain planning algorithms such as the partially ordered Hierarchical Task Network (HTN) planner that can encode rich information about the environment. Using only planner to solve the sparse reward problem by encoding information about the environment is not a viable solution. This is because the distribution of reward is not initially known. A learning algorithm is required to learn the key aspects of the environment. In addition to the information of the distribution of rewards, there are other information about the

environment that can be known and can be incorporated in the knowledge rich planning algorithm. Using merely a planner to solve the perpetual sparse reward problem is not possible as there is no clear terminal or exit condition in a perpetual sparse reward problem. Since an exit condition or a clearly defined goal state is required for a planner to operate only using planner to solve this problem is not possible.

The key inspiration of this work has been Freitas *et al.* (2014) where the authors delineate an algorithm to extract automated HTN planners from ontologies. Freitas *et al.* (2014) uses information about the agent and its environment represented semantically in the form of an ontology. An ontology is a formal description of knowledge as a collection of concepts within a domain and their relationships. Individuals (instances of objects), classes, properties, and relations, as well as limitations, rules, and axioms, must all be explicitly specified in order for such a description to be possible. The ontology was encoded in OWL (Web Ontology Language) as published in the work by Bechhofer *et al.* (2004). OWL is based on Description Logics.

This work is a novel approach of integrating reinforcement learning with planning to solve the perpetual sparse reward problem with goal-conditioned policies. The goal-conditioned policies are learnt from a planner that encodes information about the environment. This is the first work to integrate HTN with planning to solve the problem of sparse rewards. The information that is known about the environment is used to learn the unknown properties of the environment.

In this work we are exploring the improvement of solution of multi-agent systems by integrating a planner with a type of reinforcement learning called Temporal Difference (TD) learning. The improvement in the solution of this multi-agent system problem will improve the quality of solving the applications of sparse-reward based multi-agent problems. In Alam and Bobadilla (2020), the authors mention that multi-robot systems has potential to vastly improve performance in important applications

such as: search and rescue, surveillance, intrusion detection, environmental monitoring, vacuum cleaning, lawn-mowing, mine sweeping, exploration, automated farming, and painting. All of the application of multi-agent systems mentioned above are instance of two problems: multi-robot coverage problem and multi-robot assignment problem. In multi-robot coverage, a team of robot is distributed to ensure desired coverage of an area. Whereas, according to Michael *et al.* (2008), in multi-robot assignment problem, a group of robot that are equally capable of performing a set of tasks in the environment is assigned only one task at any given time based on certain constraints.

The simulation in this work is an abstraction of the persistent monitoring problem which is a type of multi-robot coverage problem. According to Alam and Bobadilla (2020), in a persistent monitoring problem, planning and execution of trajectories for multiple robots is performed to visit a known set of regions of interest continually by finding collision-free solutions to the robots trajectories.The reward in the form of bubbles is an abstraction of the value to allocating UAVs to different regions, matching drone supply to intelligence demand.

An interesting application of coverage that is tracked in the premise of this work is Chaimowicz and Kumar (2007), where a group of UAVs are used to coordinate and control a swarm of ground robots. The formation and pose of the controlled swarm of ground robots is tracked by blimps, which are UAVs. The blimp controller tracks the pose and shape of its groups to keep the robots inside its field of view. In this work it is seen that there is an incentive or a value associated with the position of blimps at certain locations in space. Like in Chaimowicz and Kumar (2007), in my work, there is a reward associated with being at certain locations in space. However, the focus of my work is not to define the control law but to improve the quality of the control law by combining planning with learning.

There has been prior work in designing a control law for multi-agent systems. Adepegba *et al.* (2016) proposes a control law in cooperation with reinforcement learning for deploying multiple autonomous agents in a two-dimensional planer area. This work demonstrates the capabilities of using reinforcement learning to develop a control law for multi-robot coverage problem. This work has inspired me to explore reinforcement learning to find solution to the multi-robot sparse rewards problem.

Georgievski and Aiello (2015) explains that in planning, only describing the goal of the problem is a necessary but not a complete solution to planning problems. Adding information about the environment in the form of boundary conditions, preconditions to tasks, modular structured task that delineate the predicates upon which a task begins and a hierarchical task structure can add to the quality of the planner's solution. Using HTN planner can add all these components to the planner. My work is not the first to use HTN planners to control multi-robot systems. There has been prior work done where multi-agent systems are controlled with HTN planners. Zeng *et al.* (2016) generates multi-agent path planning with HTN. However, this is an interesting work where learning has been combined with planning to improve the quality of solution of the controller. The learning algorithm used in this work is Temporal Difference Learning. The following section introduces Temporal Difference learning.

1.1 Temporal Difference Learning

Temporal-difference (TD) learning refers to a particular class of model-free reinforcement learning (RL) methods which learn by bootstrapping from the current estimate of the value function. Just like Monte Carlo methods, these methods sample from the environment to learn about the environment. Sutton and Barto (1987) introduced the concept of temporal difference for classical conditioning behavior. The work by Tesauro *et al.* (1995) is one of the earliest use of Temporal Difference learn-

ing to abridge the gap between real world problems and problems in simulation with regards to the temporal credit assignment problem.

In TD learning, the state value function of the MDP is given by:

$$V^{\pi}(s) = E_{a\ \pi}\left\{\sum_{t=0}^{\infty}\gamma^{t}r_{t}(a_{t})\middle| s_{0}=s\right\}$$

where r_t and γ^t represent the reward and the discount rate at time t, respectively. $V^{\pi}(s)$ satisfies the Hamilton–Jacobi–Bellman Equation:

$$V^{\pi}(s) = E_{\pi}\{r_0 + \gamma V^{\pi}(s_1)|s_0 = s\}$$

The TD learning algorithm starts with initializing a value table $V(s)$. α is the learning rate chosen. A chosen policy π is repeated to evaluate the value of state.

$V(s) \leftarrow V(s) + \alpha(r + \gamma V(s^{'}) - V(s))$

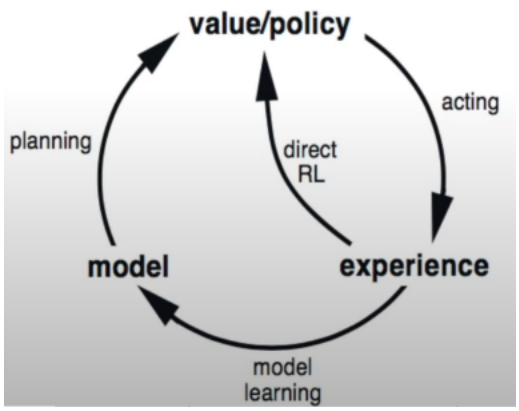

Figure 1.1: Learning With Planning, From (Sutton and Barto, 2018, Chapter 8)

Sutton and Barto (2018) introduce the concept of learning with planning in their book: Reinforcement Learning. As depicted in Fig. 1.1, after an agent takes an action following a policy π , the experience of the action can be observed by the agent and an optimal policy can be identified. The experience observed can also be used to define a state of the agent that can be used to learn a model of the environment to find an optimal policy at that state.

The same diagram in Fig. 1.1 can be used to indicate the position of the planner in the learning algorithm. However, in this work we are not learning the planner but we are using the planner for learning.

Akkaya *et al.* (2019), Savinov *et al.* (2018), Zhang *et al.* (2018) and Faust *et al.* (2018) are are some distinguished prior works where a planner has been integrated with learning. Just as in these cited works, my work uses goal-conditioned policies Kaelbling (1993). Goal conditioned policies take an input state s and a goal state s_g and find output actions or policies to navigate to the goal state. It can be quite challenging for Reinforcement Learning (RL) to learn policies to solve long-horizon tasks. However all the algorithms pertaining to goal-conditioned policies is designed to reach nearby states as a step towards the goal state.

1.2 Sparse Reward Problems

The problem of sparse-reward has been addressed using numerous approaches of reinforcement learning. The three essential approaches for solving this kind of problem are curiosity driven methods, curriculum learning and auxiliary tasks. The agent is encouraged to visit unseen states that will help the agent to identify reward tasks. A notable work in curiosity driven method is the work by Pathak *et al.* (2017), where the exploration problem is solved by encouraging the agent to explore the environment such that the agent chooses actions that reduce the error of predicting its results.

In curriculum learning, the agent is present with numerous tasks in a meaningful sequence. The tasks get more complex over time until the agent is able to solve the initial tasks.

In auxiliary learning, along with reward tasks, auxiliary tasks are used during the training process. The auxiliary tasks are not based on the main task designed in the form of a curriculum in curriculum learning. Instead, the auxiliary tasks can be

differentiated into auxiliary control and auxiliary reward prediction tasks.Jaderberg *et al.* (2016) use auxiliary tasks to address the lack of continuous reward signals.

While these approaches explore specific techniques of reinforcement learning used to address the problem of sparse rewards, this work focuses on integrating planning with reinforcement learning to address the problem of sparse rewards. Charlesworth and Montana (2020) introduces PlanGAN and integrates it with Reinforcement Learning address the problem of sparse reward with multiple goals. PlanGAN generates trajectories as plans that take the agent from its current state to the goal state.

In this work, the problem of sparse rewards is solved with goal-conditioned policies which is a novel approach of integrating learning with planning. This is the first work to integrate HTN with planning to solve the problem of sparse rewards.

1.3 Hierarchical Task Network

HTN is based on well-structured and well-conceived domain knowledge. Such knowledge is likely to contain rich information and guidance on how to solve a planning problem, thus encoding more of the solution than was envisioned for classical planning techniques. This structured and rich knowledge gives a primary advantage to HTN planners in terms of speed and scalability when applied to real-world problems and compared to their counterparts in classical world.

According to Alford *et al.* (2009) there are two types of classical planners: domain-independent and domain-configurable. HTN comes under the category of domain-configurable planner. Unlike domain-independent planners, domain-configurable planners such as HTN utilizes the domain-specific planning knowledge in the form of control rules or HTN methods.

SHOP is an HTN-based planner that shows efficient performance even on complex problems, but at the expense of providing well-written and possibly algorithmic-like

domain knowledge.

HTN planners have 4 essential descriptive units. They are as follows:

State: A state is description of the different situations in the planning process.

Task: A task is a description of an activity to perform. There are two types of tasks: primitive task and compound task.

Operator: An operator is a parameterized description of what a basic action does. Each operator describes how state variables are updated with preconditions and effects. Actions are operators with arguments.

Method: A method is a parameterized description of ways to perform a compound task by performing a collection of sub-tasks.

HTN planners take advantage of the least-commitment strategy. According to Weld (1994), HTN planners need to make two decisions on constraints. The first decision is on the use of for binding variables and the second decision is on the ordering tasks in a task network. In least-commitment strategy, the ordering of tasks and variable bindings are deferred until a decision of a solution is forced.

A Partial-order planning (POP) is a approach of least-commitment strategy planner which maintains partial ordering between actions. The planner commits ordering between actions only when forced by constraints. In contrast to partial-order planning (POP), in total-order planning, these is a fixed order of actions. In problems where some sequence of actions are required to achieve a goal, a partial-order plan specifies all actions that need to be taken, but specifies an ordering between actions only when necessary.

The advantage of HTN planners is primarily their sophisticated knowledge representation and reasoning capabilities. Ghallab *et al.* (2004) discusses the rationale for calling the HTN planner an "information-rich" planner. Kambhampati (1995) claims that the primary advantage of using HTN planning is the flexibility HTN planning

can provide the user in the type of solution it can generate. Nau *et al.* (1998), Lekavỳ and Návrat (2007) and Erol (1995) has discussed about the expressiveness of HTN planners as an advantage of this type of planning.

Georgievski and Aiello (2015) has identified three categories of properties that a HTN planner has, which aid its expressive power. The first category of property covers the system of first-order logic with helps in ensuring logical connectives. HTN can use conjunction ($\wedge$), disjunction ($\vee$), negation ($\neg$) , universal quantifier ($\forall$) and existential quantifiers ($\exists$). Logical connectors or quantifiers can be applied on preconditions and effects of operators and method.

The second and most important category of properties is the quality constraints. These constraints makes HTN both more descriptive and flexible in terms of goal. With regards to the quality of planner the mention-able properties are: Typing, Extended goals and Preferences. HTN enables expressing types of object in the form of a type hierarchy. This aspect of HTN enables the ease of conversion of HTN to ontologies and vice versa. Extended goals helps in expressing a planning objective in a way such that the satisfaction of the objective could occur at any intermittent point on the whole trajectory of the solution, or it could occur in the final state. Preference is a condition in the solution trajectory that a user would prefer to be satisfied in the solution of the problem. However, a solution that does not include the preferred constraint is still an acceptable solution to the problem.

The third category of property that makes HTN expressive are the resource and time constraints. These constraint can be used in encoding real-world resource and time limitations in the planner. Compared to other classical planning algorithms, the structured format and rich knowledge representation format of HTN gives a primary advantage to HTN planners in terms of scalability and speed when applied to real-world problems.

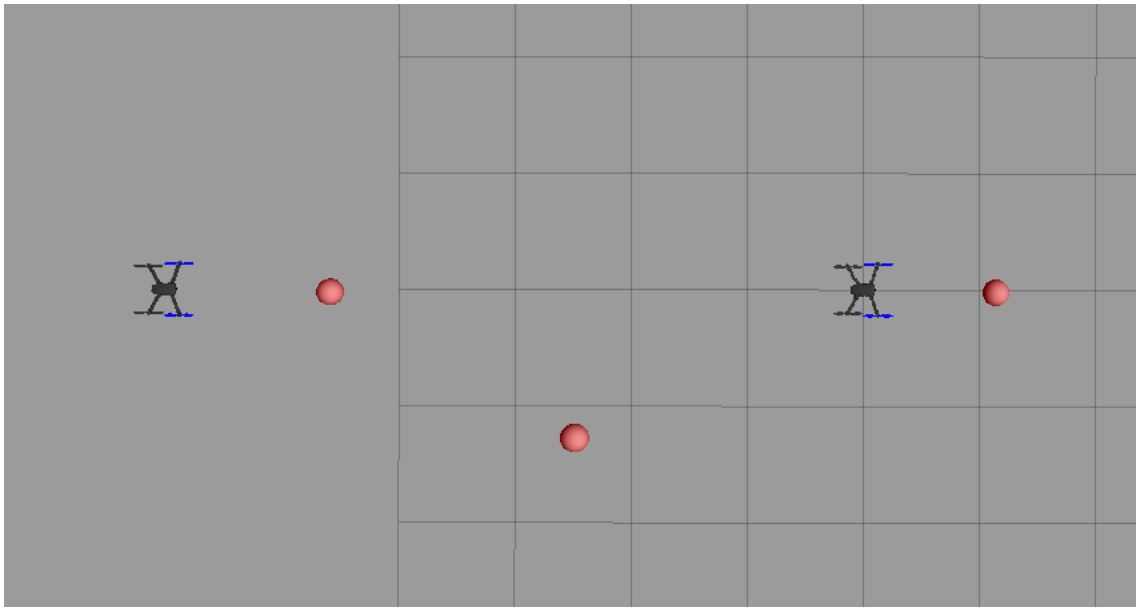

Figure 2.1: Top-view Of The Simulation Environment

The simulation of this algorithm has been performed in the ROS-based (Quigley *et al.* (2009)) physics engine called gazebo simulator (Koenig and Howard (2004)). ROS is an open source framework of robotic software tools that are used in robotic systems. ROS allows for controlling robotic systems with an architecture similar to a network with nodes. The ROS nodes can run as individual entities and can performs computation and communicate information with other ROS nodes. The communication between these ROS nodes takes place with a channel of streaming data known as rostopic. Each ROS node can either subscribe to data from these rostopics or can publish data from these rostopics. The data that is transferred between the nodes are called ROS messages. In our simulation, there are 4 scripts that run as separate ROS nodes.

Gazebo is a robotic simulation that allows to test the robotic system in a real-world-like physical environments. Gazebo simulator comes with a physics engine. It integrates directly with ROS, making simulation of ROS-based robotic systems seamless. Gazebo allows for the creation of custom worlds and robots, allowing for a plethora of simulation environment and robotic systems. ROS and gazebo makes the basic framework of the robotic simulation in this work.

The robotic agents in this work are unmanned aerial vehicles (UAVs) in simulation. PX4 (Meier *et al.* (2015)) flight stack has been used to simulate the UAVs in this work. PX4 is an open-source flight control system for aircraft, ground vehicles, and watercraft. PX4 allows the UAV to have low-level rate control as well as higher-level velocity and position control. Sensor support, flight logging, and the state estimator are all included in the PX4 flight stack. The MAVlink communication protocol enables the UAVs in the PX4 flight stack to communicate with ground stations or the simulation environment.

The initial set up of the simulation environment has been performed using the OpenUAV (Schmittle *et al.* (2018)) platform. OpenUAV is a multi-robot design and testing bed in simulation. It is based on the based on the ROS, gazebo and PX4 flight stack.

In this experimental setup, we have chosen a 3-D space with 8 positions that a UAV can assume. In combination to these 8 positions described by three coordinate values each for the x-coordinate, y-coordinate and the z-coordinate respectively, an few other parameters are used to describe a state of the UAV which will be explained in the subsequent paragraphs. In this set-up there are 3 reward bubbles situated at 3 of these 8 assumable positions.

If the UAV is at a distance less than 1 m from the reward bubble, the reward bubble pops. Once the bubble pops, the bubble does not relapse for the next 3

seconds and until there is no UAV present anywhere at a radius of less than 1 m from the UAV. The bubble relapses and a new reward is seeded at the same location in the environment after 3 seconds from when the reward bubble popped, provided there is no UAV in the vicinity of 1 m.

The UAV also needs to be aware of the presence of other UAVs in the environment. In this regard the proximity and the direction of the presence of the other UAV is the vicinity is encoded with 5 numbers. Figure 2.2 shows the direction encoding value of the presence of other UAV in the vicinity.

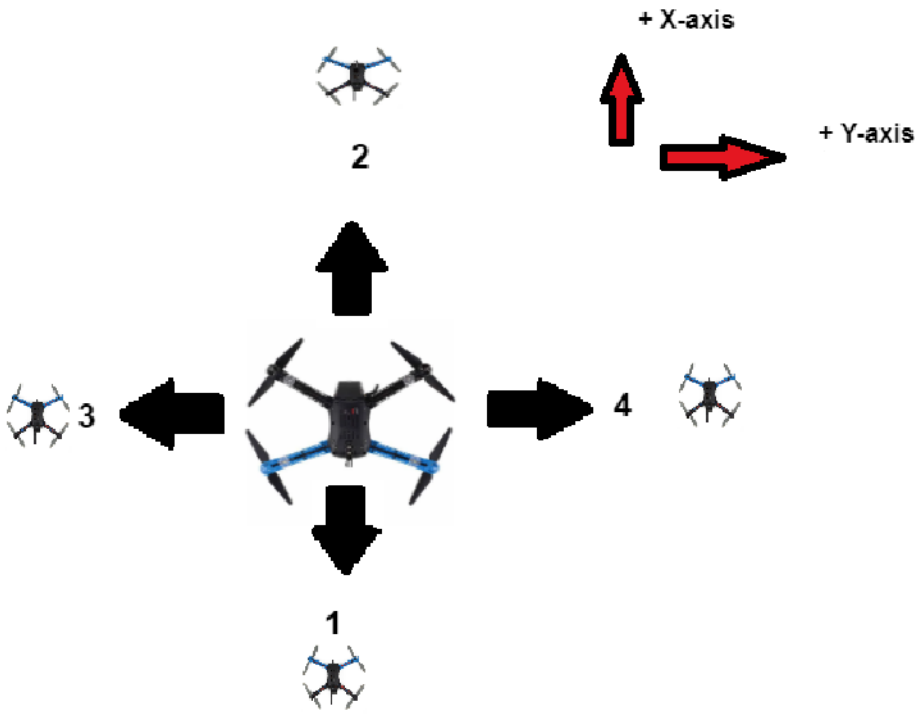

Figure 2.2: Animated Direction Encoding

The following table enlists the encoded values and the direction in which another UAV is present:

The UAVs need to be aware of how long the reward bubble shall take to relapse.

Table 2.1: Direction Encoding To Track Other UAV.

Direction encoding values	Direction of another UAV
1	negative x direction
2	positive x direction
3	negative y direction
4	positive y direction
0	no other UAV in the proximity

This can be achieved by encoding how long the reward bubble is gone in seconds. The time for which each of the rewards has popped can assume 3 values: 0, 2, and 3. The following table encodes how long the rewards bubbles are gone as discrete integer timestamps. This encoded value of time will be referred to in this literature as $\text{time}_{\text{reward}}$ followed by a number to indicate which reward bubble is being referred to.

Table 2.2: Time Encoding Of The Collected Reward .

Direction encoding values	Direction of another UAV
0	reward collected recently
2	reward collected 2 second ago
3	reward is available for collection again

The state S of the UAV is defines as follows:

S = **{x-coordinate of UAV, y-coordinate of UAV, z-coordinate of UAV, $\text{time}_{\text{reward1}}$, $\text{time}_{\text{reward2}}$, $\text{time}_{\text{reward3}}$, Direction encoding values }**

There are seven variable parameters that are used to encode the state of a UAV.

Each of the fields corresponding to the $\text{time}_{\text{reward}}$ can assume 3 values. There are a total of 8 state locations, 27 combination of $\text{time}_{\text{reward}}$ values and 5 possible direction encoding of other UAVs. So there is a total of 1080 states in this simulation.

METHODOLOGY

As introduced in the previous chapter, Temporal Difference (TD) learning is a model-free reinforcement learning process. I have chosen Temporal Difference (TD) learning over model-based learning approaches to emulate real-world-like condition in simulation. Like in real-world, the agent in simulation learns about the environment and the change in environment as it explores the environment. As discussed in the chapter: simulation environment, the agent learns about the environment and decides its state through constantly streaming information from the data channels called a rostopic. These can be thought of as sensory input data in real-world simulation. The agents in this work are UAVs in simulation. In this chapter, the terms UAV and agent have been used interchangeably.

In each new state that the agent visits, a customized action list is generated. The boundary conditions of the simulation environment is incorporated by this action list of action. The action list for a state will not have actions that will change the current state to states that are not permissible. For example, if the agent is at a location (-1,0,3) and we do not consider any positions where the value of x is lesser than -1, then our action list will eliminate the action that may result in landing at a state where the value of x is lesser than -1.

```
** result = [('move_left', <pyhop.Goal instance at 0x7fe7ee022c30>), ('move_back
ward', <pyhop.Goal instance at 0x7fe7ee022c30>), ('move_backward', <pyhop.Goal i
nstance at 0x7fe7ee022c30>), ('move_backward', <pyhop.Goal instance at 0x7fe7ee0
22c30>)]
```

Figure 3.1: Example Of A Generated Plan

This work uses Pyhop (Nau (2013)) which is a simple HTN planner written in Python to generate plans. Pyhop's planning algorithm is similar to the SHOP category of HTN planners. There are however some differences between Pyhop and SHOP like HTN planner. The biggest advantage of Pyhop is the easy of integrating it with other computer programs especially in a python development environment. Unlike logical propositions, Pyhop represents the state of the world using ordinary variable bindings instead of logical propositions.

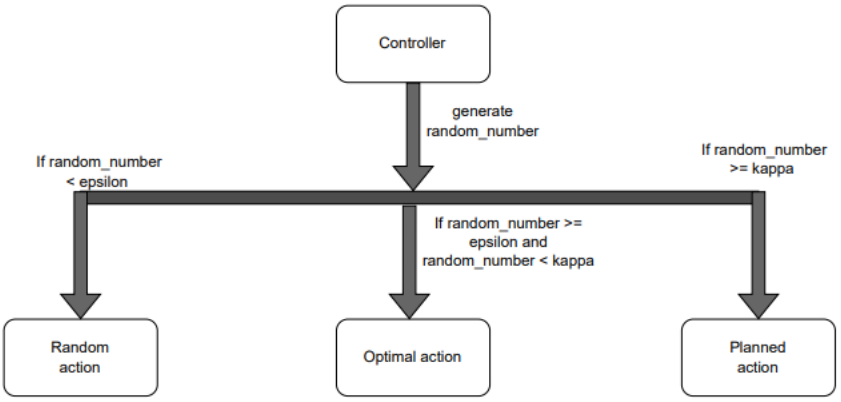

Figure 3.2: Action Decision Tree

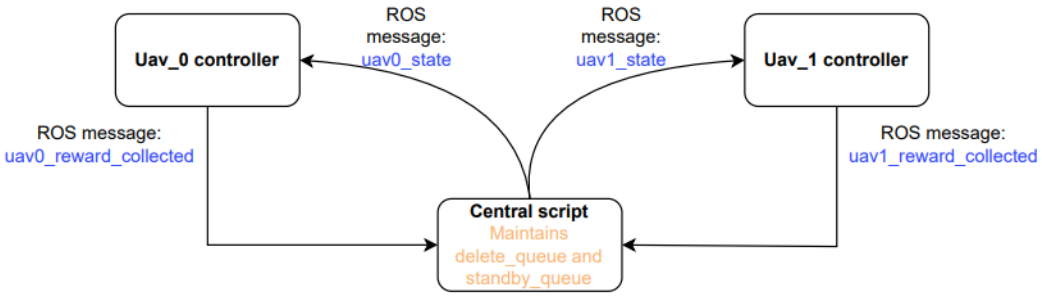

Figure 3.3: Mechanism To Control Bubble Popping And Bubble Relapse

In this work, we are using a specific type of algorithm of Temporal Difference (TD) learning called Q-learning. Two main algorithms are being explored in this

Algorithm 1: Q-learning

while *node Q-LEARNING has not shut down* **do**

s ⟵ *state of UAV* ϵ ⟵ *explorating-exploitation factor*;

state_dictionary ⟵ *map of visited states and action-qvalue pairs* ;

optimal_action ⟵ sort(state_dictionary) at s ;

pruned_action_list ⟵ prune_action() at s and apply boundary conditions;

random_action ⟵ randomly choose action from pruned_action_list ;

if $\epsilon \leq random.random()$ **then**
  action_to_perform ⟵ random_action;

else
  action_to_perform ⟵ optimal_action;

success_status, next_state ⟵ execute_action();

if *next_state not in state_dictionary* **then**

  next_state_prunned_action_list ⟵ prune_action() at next_state ;

  create_action_qvalue_pair at next_state ;

  Add next_state to state_dictionary

else
  pass

reward ⟵ get_reward for state, action, next_state ;

update_cumulative_reward() using current reward ;

q_old ⟵ from state_dict at s for action_to_perform ;

q_value_pair ⟵ from state_dict at s q_value ⟵

$((1-\alpha) * q\_old) + \alpha * (reward + \gamma * max(state\_dict[next\_state].values()))$

update q_value_pair with q_value ;

update state_dict with q_value_pair

Algorithm 2: Q-learning with planning

```
while node Q-LEARNING has not shut down do
    s ⟵ state of UAV ϵ ⟵ explorating-exploitation factor;
    κ ⟵ planning-exploitation factor;
    state_dictionary ⟵ map of visited states and action-qvalue pairs ;
    optimal_action ⟵ sort(state_dictionary) at s ;
    pruned_action_list ⟵ prune_action() at s and apply boundary conditions;
    random_action ⟵ randomly choose action from pruned_action_list ;
    random_number ⟵ random.random() ;
    if ϵ ≤ random_number then
        action_to_perform ⟵ random_action;
    else
        if ϵ ≥ random_number and κ ≤ random_number then
            action_to_perform ⟵ optimal_action;
        else
            action_to_perform ⟵ planning_action;
    success_status, next_state ⟵ execute_action();
    if next_state not in state_dictionary then
        next_state_prunned_action_list ⟵ prune_action() at next_state ;
        create_action_qvalue_pair at next_state ;
        Add next_state to state_dictionary
    else
        pass
    reward ⟵ get_reward for state, action, next_state ;
```

while *node Q-LEARNING has not shut down* **do**

update_cumulative_reward() using current reward ;

q_old ⟵ from state_dict at s for action_to_perform ;

q_value_pair ⟵ from state_dict at s q_value ⟵

$((1-\alpha)*q\_old)+\alpha*(reward+\gamma*max(state\_dict[next\_state].values()))$

update q_value_pair with q_value;

update state_dict with q_value_pair

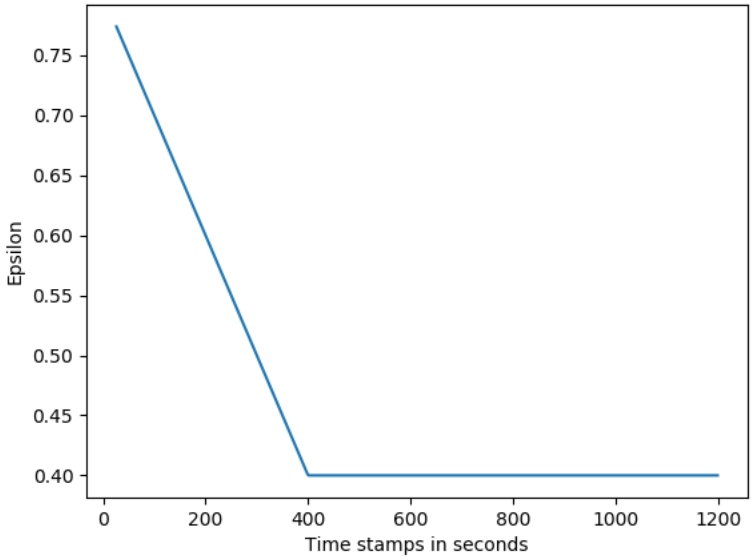

Figure 3.4: Epsilon (ϵ_{exp}) VS Time Stamp For Temporal Difference Learning

work- Algorithm1: Q-learning, Algorithm2: Q-learning with planning. In both the algorithms, the state of the agent, S is defined according to the state introduced in the chapter: Simulation Environment. Each state is a vector with 7 fields. The term Q-learning and Temporal Difference learning will be used interchangeable in this literature. As mentioned in the introduction section, the Value function, V(S) is learned during the execution of Temporal Difference learning algorithm. In Q-learning, each state-action pair has a q-value associated with it.In Q-learning, the Q

function, Q(s,a) is learned. The Q function varies with both- state and action. Each state has as many q values as the number of permissible action from that state.3.1 is the used to update the Q function upon executing each action in the simulation. The following section describes Q-learning in details.

3.1 Q-Learning

At every state the agent visits, the agent can choose two types of action. Either the agent can choose the optimal action from its previous experiences of visiting that state, or the agent can choose a random action from the list of permissible action from that state. The exploration-exploitation factor ϵ, is used as a threshold to compare a randomly sampled number and choose the optimal action or the random action.

A *state dictionary* is maintained to enlist a mapping between all the visited states and the *action q-value pair*. When a new state is visited, a new entry is made in the *state dictionary*. A *state dictionary* is similar to a Q-table, which is commonly used in Q-learning. The *state dictionary* is a python dictionary that is used to maintain the same relationship between states and q-values like in a Q-table. The *action q-value pair* is also a python dictionary. This is referred as *q-value pair* in the Algorithm1 and Algorithm2. Upon visiting a new state, the *action q-value pair* is instantiated a customized action list as discussed in the beginning of this chapter. This customized action list is referred to as the *pruned_action_list* in Algorithm1 and Algorithm2.

After the chosen action is executed, the cumulative reward is updated irrespective of whether a reward was collected or not after execution an action. The current reward collected is discounted by a factor *gamma* which is commonly referred as the discount factor across literature. The discount factor *gamma* used in this work is a time varying number. The discount factor is designed such that rewards collected initially in the simulation are given greater weightage than rewards collected later in the simulation.

The cumulative reward is updated by the function, update_cumulative_reward(). It uses the following formula to update the cumulative reward:
$reward_{cumulative} = reward_{previous} + \gamma^{step} reward$, where reward is the reward collected at the current time-step.

After the reward is collected, the *action q-value pair* is updated with a new q-value corresponding to the action that was taken at the previous state. This algorithm uses a weightage factor α to combine the previous q-value corresponding the chosen action at the state, with the newly calculated q-value after collecting the reward. The q-value is calculated using the following formula:

$$Q(s,a) = (1-\alpha) * Q(s,a)_{old} + \alpha * (reward + \gamma * max_a Q'(s',a')) \tag{3.1}$$

Here Q(s,a) stands for the q-value at state S for action a. S' stands for the next state. Q'(s') stands for all q-values at state S'. max_a prefix of Q'(s',a') denotes the action corresponding to the maximum Q value at state S'.

3.2 Q-Learning With Planning

In Q-learning with planning, all the steps remain the same at in the Q-learning algorithm. A new factor κ, the planning-exploration factor has been introduced in this algorithm. Figure 3.2 depicts the action taken based on the value of the random number sampled. In this algorithm, there are three possible categories of action outcomes. Based on the value of the random number and the threshold factors, ϵ and κ, a random action can be chosen, an optimal action can be chosen or a action suggested by the planner can be chosen. For Q-learning with planning, I have chosen a constant value of 0.1 and 0.3 for ϵ and κ respectively. Hence, the probability of the planner being chosen to indicate the optimal action is 0.7.

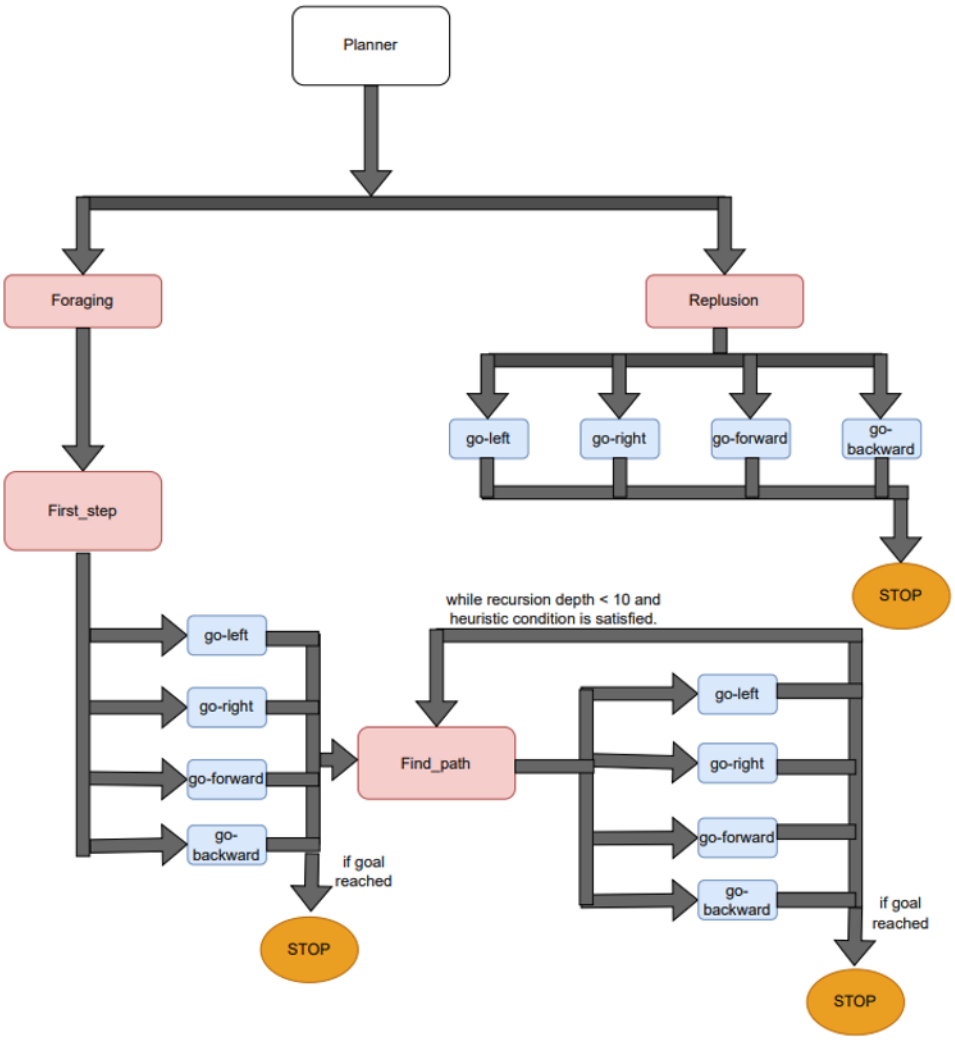

Figure 3.5: Implemented Hierarchical Task Planner

3.3 HTN Planner Design

Figure 3.5 depicts the flow diagram of the HTN planner. Since we are using Pyhop for designing this HTN planner, the traversal ordering of this planner is fully-ordered. This planner implements depth-first search with backtracking to reach the goal. Each of the levels of the planner are tasks. There are two main tasks defined in our HTN planner- Foraging and Repulsion. These tasks have different goals. The planner first performs a depth-first search inside foraging to find if there is a reward available

nearby. If a goal is not reachable within a threshold number of steps or a horizon, then the planner will try to execute task called repulsion.

Under foraging, the planner first identifies all the nearby available rewards and chooses the nearest reward. If there is a way of reaching the closest reward following a heuristic defined shortest path, then the planner will find a plan to execute a set of actions to reach the closest reward. The closest reward is the goal of the planner.

If the planner is unable to find such a path because of the presence of another agent in the path or because of large number of steps to reach the reward or goal, then the planner will abort the execution of the foraging task. The planner will then execute repulsion, where the agent moves away from the presence of another agent that has stymied the path of the agent. In a cluttered environment like in our simulation, repulsion task helps in uncluttering the vicinity of the agents. The repulsion task checks if any of the actions in the list of permissible actions will lead to a greater distance between the agents than the previous distance between them. Under the task repulsion, the planner will reach a goal state the moment an action in repulsion is identified that succeeds in achieving this goal criteria.

Tasks under Foraging has two hierarchies- First_step and Find_path. The first_step is a task that can have only one action. The first_step helps avoid collision between agents during planning. The first_step considers the next state of the other agent during planning. The first step is taken such that is does not visit the current position and the next position of the other agent. This is crucial since the Q-learning with planning algorithm only executes the first action from the sequence of action produced by the planner. The Find_path task helps us check how many steps it will take to reach the nearest goal or reward point. If the number of action required to reach the goal exceeds a threshold or does not follow the heuristic-defined path, then the agent exits executing the Foraging task. Find_path helps in ensuring that the goal is

reachable following a specific trajectory within a the threshold number of action-steps.

3.4 Simulation Mechanisms

Figure 3.3 depicts the latent mechanism to maintain and update states. There are two separate controllers running on the two agents. These controllers execute the Q-learning and the Q-learning with planning algorithm. These controllers subscribe to and publish to rostopics to learn about the agent's state and execute actions. The central script is the action server. This script handles the laws of the environment. The action server ensures that: the reward are popped and replenished based on the acceptance radius of the bubbles, the reward values are assigned to the correct agents, the state of the agents are updated based on the actions that were executed by the agents. After a reward is collected, the reward is not available for 3 seconds. This is updated to the central script by the $uav0\_reward\_collected$ and $uav1\_reward\_collected$ rostopics. The rostopics uav_0 and uav_1 are published by the central script or the action server. Both agents can access information on each others state by subscribing to the uav_0 and uav_1 topics.

To execute dynamically popping bubbles and adding them after a fixed amount of time in simulation, two services have been written in ROS. These services deletes and adds models in the gazebo simulator. The delete_model service is called when the agent is at the acceptance radius of the reward bubble. The add_model service is called when all agents are outside the acceptance radius of the reward and the reward was last collected at least 3 seconds ago.

To preserve the information of each reward bubble's location, a python dictionary called a sphere_dict is used. This dictionary has the location of the reward bubbles in a tuple as key and the name of the sphere as the value. This dictionary is used to instantiate the initial simulation environment. This dictionary is also used to look

up the location of the popped sphere while adding it back. A copy of this dictionary is used to create a map of all the available rewards and unavailable rewards. When a reward is collected, the entry from a copy of the sphere_dict, associated with the collected reward is removed. When the reward is replenished again, a new entry is made to this copy of the dictionary. The copy of sphere_dict helps in bookkeeping the availability of rewards in the environment. delete_model and add_model calls for correctly tracking the model of the reward bubbles in the environment. Calling these services at the two time will lead to error in the simulation.

To reduce the computation time of executing the dynamics of environment, a mechanism to only consider reward bubbles in the vicinity of the agent is use. Instead of checking if the agent is in the acceptance radius of any reward bubble,the search space is reduced by considering bubbles on in the vicinity of the agent. A binary search algorithm runs over a list created from the key entries of sphere_dict. The key entries of state_dict are tuples that encode the co-ordinates of the bubbles. This list is sorted based on the first entry of each tuple. The first entry of each tuple is the x-coordinate of the location of the reward bubble. A binary search algorithm runs twice to identify the lower limit and the upper limit within the threshold of consideration around the position of the agent. Only bubbles within the threshold of consideration will be considered for checking if the bubbles are within the acceptance radius of the reward. Unlike linear search over all the reward bubbles which has a time complexity of $\mathbf{O}(n)$, the binary-search-based algorithm has a time complexity of $\mathbf{O}(\log n)$. Since the position of the agent updates quickly in this simulation, reduction of the time complexity for the proximity check of agents from the reward location is a significant reduction in the processing time of latent mechanisms during simulation.

3.5 Robustness To Uncertainties

Using the PX4 flight stack with gazebo comes with uncertainties. The UAVs are often not precisely in their expected states. After choosing the optimal action or the random action at any given state, target way-points are generated as position target values. A Proportional-Integral controller is used to reduce the error between the current state and the target way-point and makes the UAV move towards the target way-point. A tolerance of 0.1 m has been provided in the simulation environment between the way-points and the position of the UAV. So, the UAV will still be considered to be situated at the target way-point when the difference in the value of the position of the UAV and the target way-point can be upto 0.1 m. The pickup radius of the reward bubble is less than 1 m. So when the UAV is not in the location of the reward but is at a target way-point that is 1 m away from it, there is a chance that the UAV still accepts the reward at this position as there is a tolerance of 0.1 m at the target way-point and the UAV can be closer to the reward than 1.

For example, the target way-point is (-2, 1, 2). The UAV can still pick up the reward at (-2, 0, 2) if the position of the UAV is (-2, 0.95, 2) instead of the target way-point. This is because the UAV's position is closer to the reward than 1 m. Meanwhile, the UAV will still be considered at the target way-point of (-2, 1, 2) even though it is actually at position (-2, 0.95, 2) because a tolerance of 0.1 has been provided in the simulation environment between the way-points and the position of the UAV. While most of the time, a reward will be collected when the UAV is at the location of (-2, 0, 2), rewards can also be occasionally collected from ways-point positions that are 1 m away from the location of the reward.

These uncertainties of actions in this environment emulates the uncertainties of actions of robots in real world. Q-learning uses the distribution of q-values to identify

optimal actions at a state. While uncertainties in actions can lead to erroneous q-values associated with certain state-action pair, these erroneous q-values will increase the chance of choosing the state-action pair associated with these aberrant actions resulting in more sampling of the state-action pair. This will result in updating the q-value of the state-action pair frequently. Thus the error due to uncertainties are reduced in this algorithm eventually in this simulation.

RESULTS

To track the evolution of the learning algorithm, I have gauged two properties of the learning algorithm: the number of states visited over time and the cumulative reward collected over time. The quality of the solution of the two algorithms being compared is tracked with these properties. An algorithm with higher value of cumulative reward will indicate that it has been able to find the locations of the reward faster and has been able to learn the temporal dynamics of the reward bubbles early on in the simulation. A lower value of cumulative reward will indicate that the algorithm requires a longer time to learn about the environment. An algorithm that takes higher number of states explored will indicate that more states were explored to find the optimal policies than in an algorithm with fewer states visited and the same amount of cumulative reward collected.

Since the trajectory taken by the agent in this simulation varies with the starting point of the agents in the simulation, the quality of learning of the environment also varies with the starting point of the agents in the simulation. Hence 60 simulations were run to observe the nature of the two algorithms with a large sample size. Two sets of simulations were run: Simulation of Q-learning without planning, Simulation of Q-learning with planning. In Q-learning with planning, ϵ and κ were chosen such that there is 70% probability of choosing an action indicated by the planner. For each of the set of simulations, 30 simulations were performed by changing the position of the agents. A total of 6 different starting points where set for the agents from which a pair of starting points are selected one by each agent. A total of 15 combinations of starting position of the agents are present. Figures 4.1, 4.2, 4.3, 4.4, 4.5, and 4.6

tracks the cumulative reward collected by UAV_0 and UAV_1 during the 60 simulations with the starting position of UAV_0 and UAV_1 changed in each simulation.

Figures 4.1, 4.2 and 4.3 tracks the cumulative reward collected by UAV_0, UAV_1 and a combination of both the agents during Q-learning without planning. Figures 4.4,4.5 and 4.6 tracks the cumulative reward collected by UAV_0, UAV_1 and a combination of both the agents during Q-learning with planning.

Figures 4.7, 4.8, shows the number of states visited by Uav_0 and Uav_1 in Q-learning without planning. Figures 4.9, 4.10 shows the number of states visited by Uav_0 and Uav_1 in Q-learning with planning.

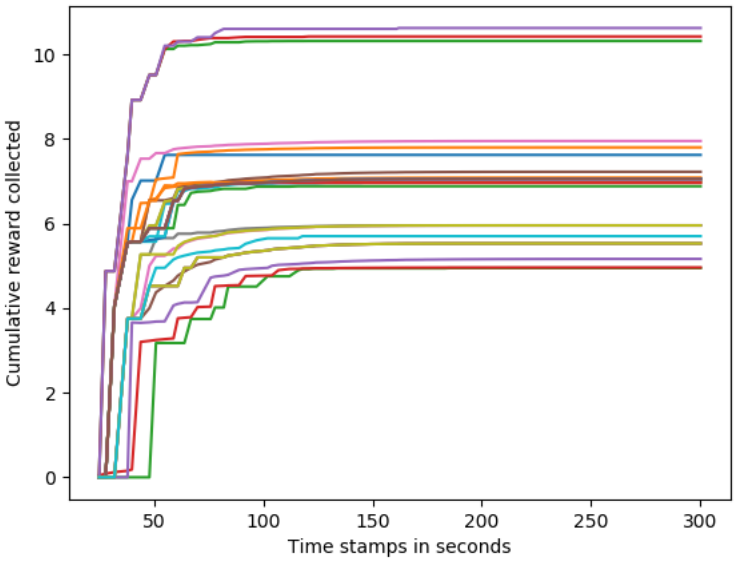

Figure 4.1: Cumulative Reward Collected VS Time Stamp For UAV_0 In Q-learning Without Planning

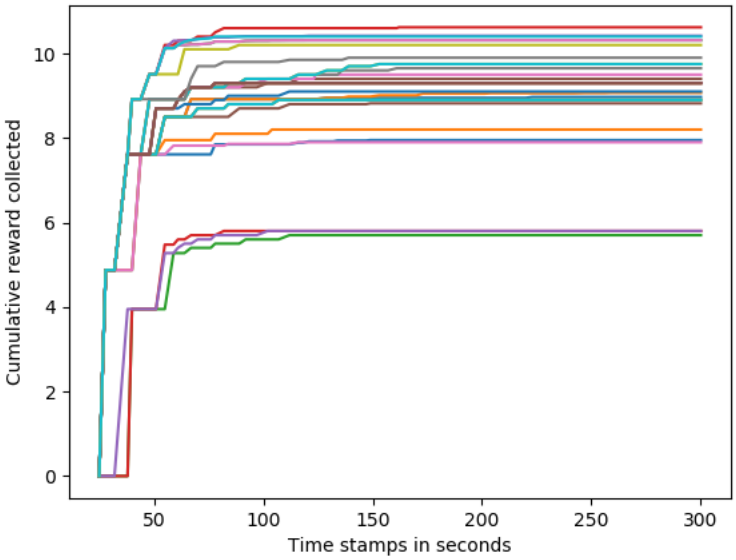

Figure 4.2: Cumulative Reward Collected VS Time Stamp For UAV_1 In Q-learning Without Planning

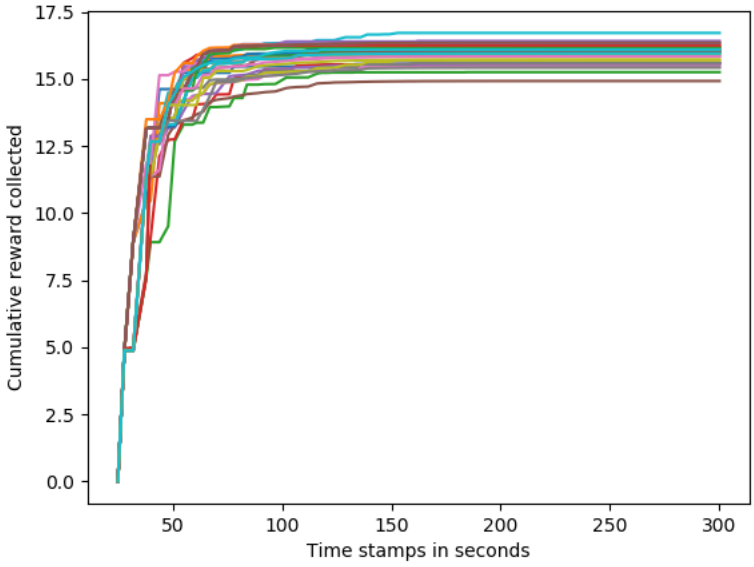

Figure 4.3: Cumulative Reward Collected VS Time Stamp For Sum Of Cumulative Reward Collected For UAV_0 And UAV_1 In Q-learning Without Planning

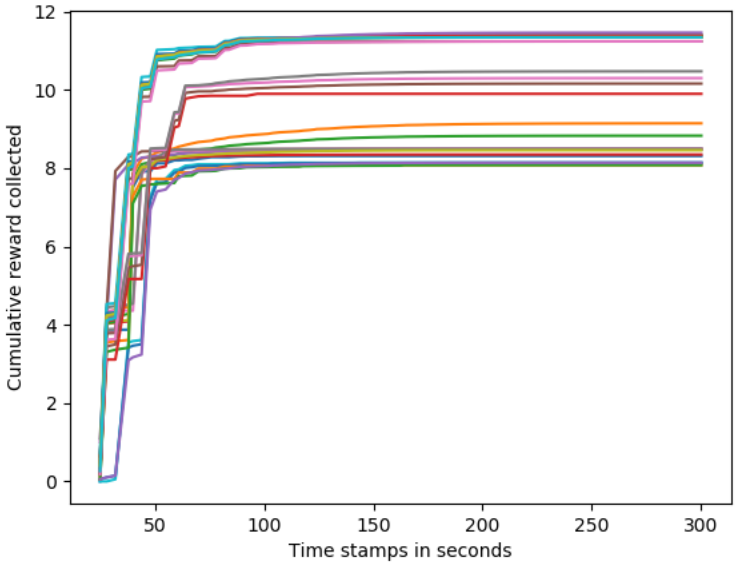

Figure 4.4: Cumulative Reward Collected VS Time Stamp For UAV_0 In Q-learning With Planning

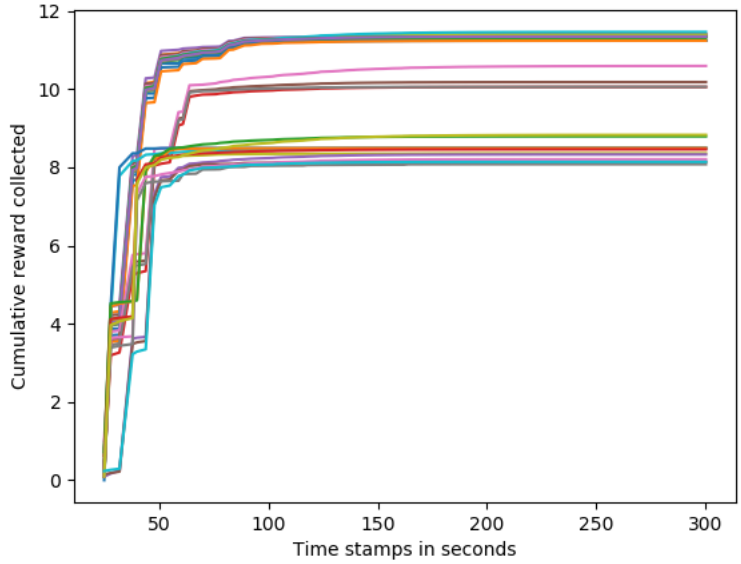

Figure 4.5: Cumulative Reward Collected VS Time Stamp For UAV_1 In Q-learning With Planning

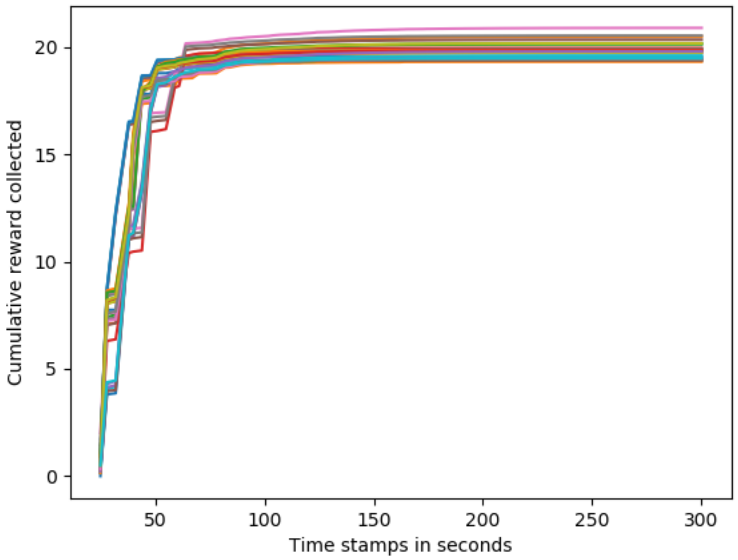

Figure 4.6: Cumulative Reward Collected VS Time Stamp For Sum Of Cumulative Reward Collected For UAV_0 And UAV_1 In Q-learning With Planning

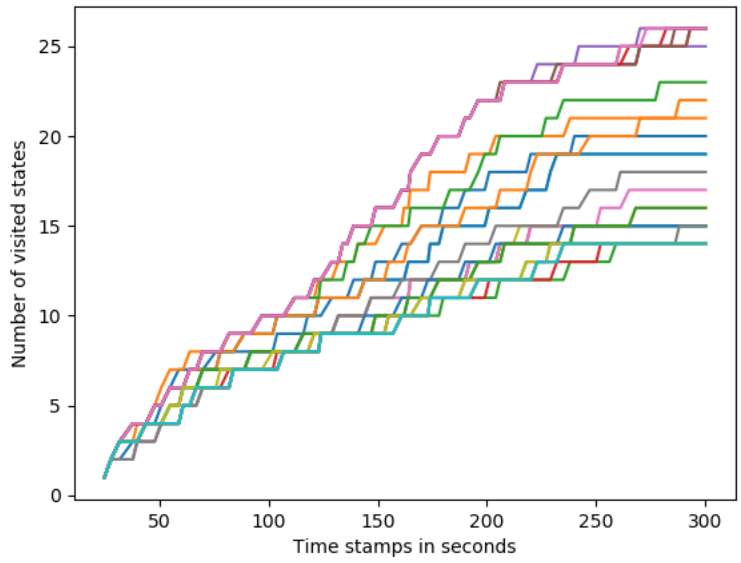

Figure 4.7: Number Of Visited States VS Time Stamp For UAV_0 In Q-learning Without Planning

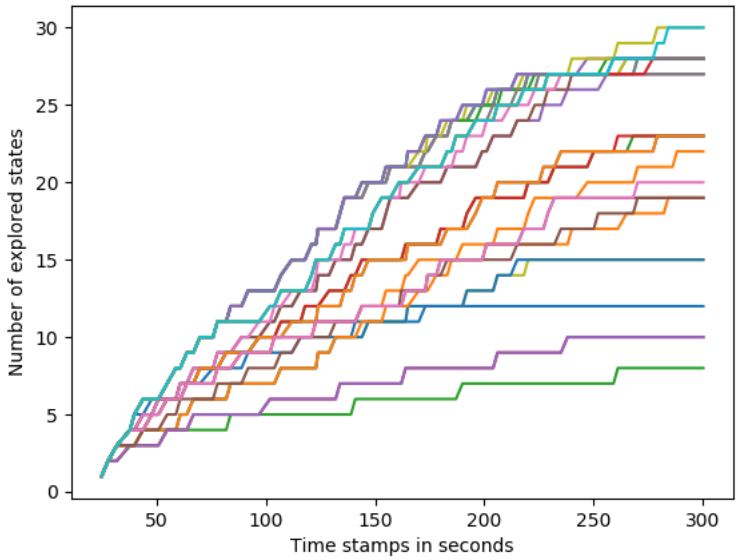

Figure 4.8: Number Of Visited States VS Time Stamp For UAV_1 In Q-learning Without Planning

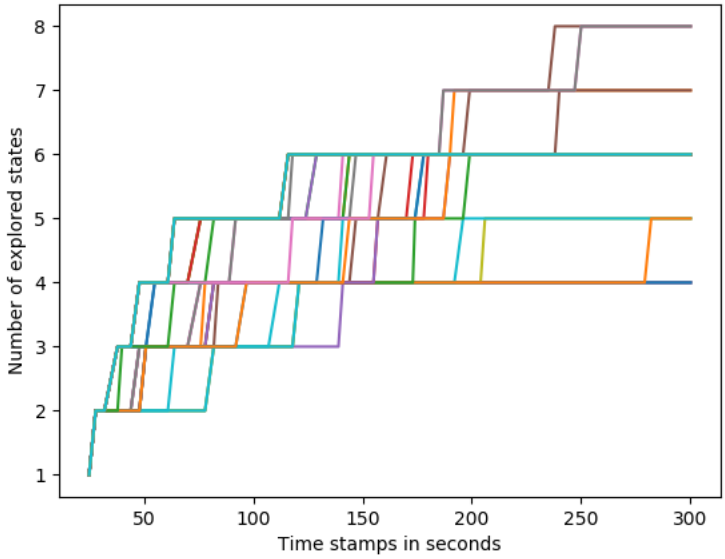

Figure 4.9: Number Of Visited States VS Time Stamp For UAV_0 In Q-learning With Planning

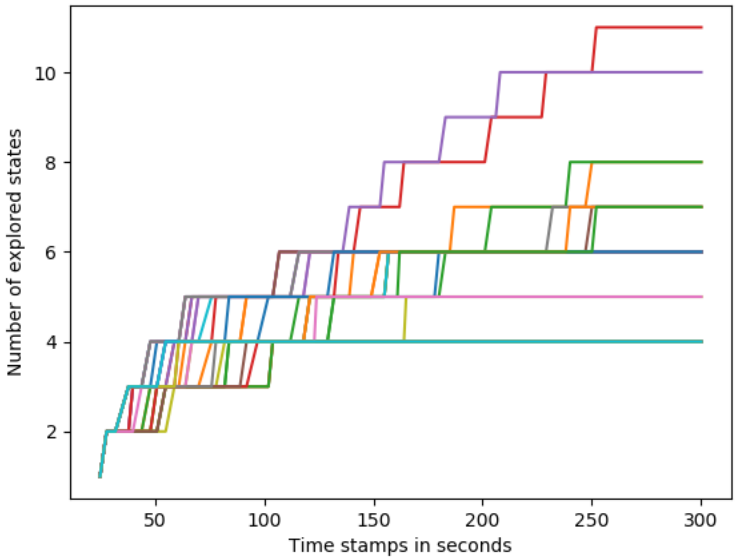

Figure 4.10: Number Of Visited States VS Time Stamp For UAV_1 In Q-learning With Planning

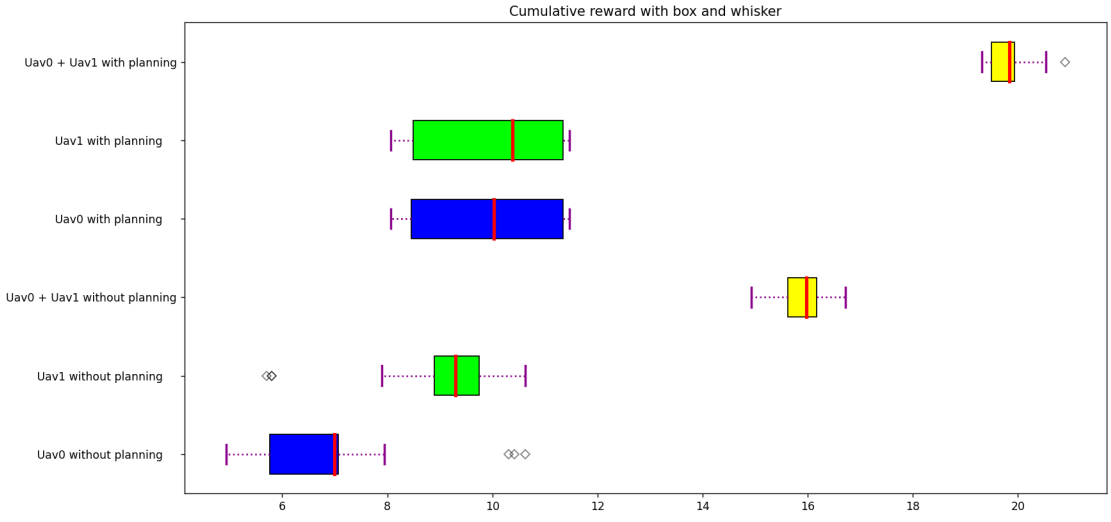

Figure 4.11: Cumulative Reward Collected For 30 Simulation Each With Q-learning And With Q-learning With Planning

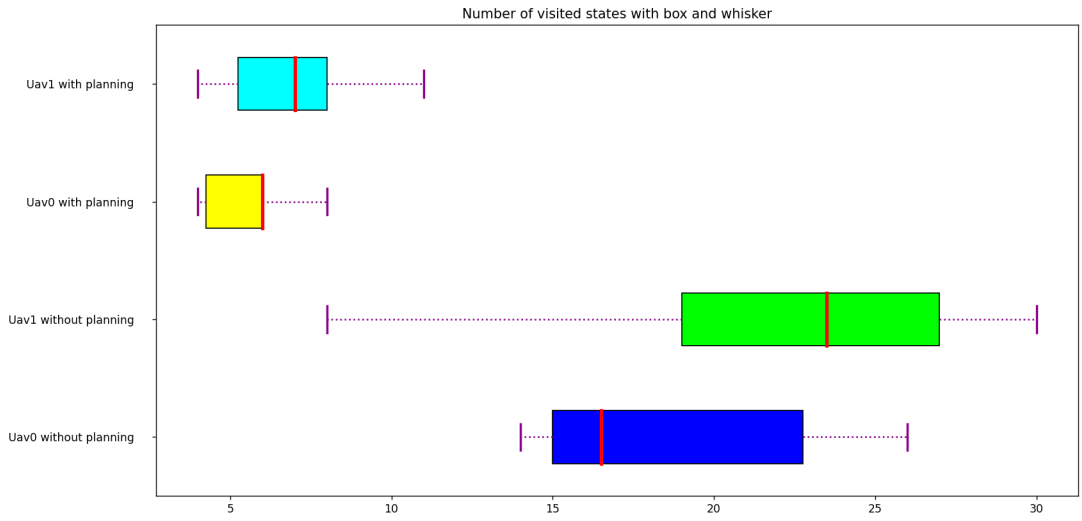

Figure 4.12: Number Of Explored States For 30 Simulation Each With Q-learning And With Q-learning With Planning

DISCUSSION

30 simulations were performed for the Q-learning algorithm without planning. Figures 4.1, 4.2 and 4.3 shows the change in the cumulative reward collected versus time for Q-learning without planning. The sum of cumulative reward of both the agents are considered for each simulation. It is observed that the mean of the sum of cumulative reward collected is around 15 in the case of Q-learning without planning.

Figures 4.4, 4.5 and 4.6 shows the changes in the cumulative reward collected versus time for Q-learning with planning. Like in the simulations of Q-learning without planning, the sum of cumulative reward of both the agents are calculated for each simulation. It is observed that the mean of the sum of cumulative reward collected is around 19 in the case of Q-learning with planning.

Figure 4.11 compares the cumulative rewards collected by each agent in both the algorithms. It is observed from Figures 4.3, 4.6 and 4.11 that the mean of the sum of the cumulative rewards collected in Q-learning with planning is greater than the cumulative rewards collected in Q-learning without planning. The ranges of the sum of the cumulative rewards collected in the two algorithms do not coincide. The range of the cumulative rewards collected in the Q-learning without planning algorithm is between 14.5 and 17.3. Whereas, the range of the cumulative reward collected in the Q-learning with planning algorithm is between 19 and 21. Thus the range of the cumulative reward collected in Q-learning with planning is more constricted in comparison to Q-learning without planning. Thus the Q-learning with planning algorithm is less affected by the changes in the starting positions of the agents in the simulation that the Q-learning without planning algorithm.

Figures 4.7, 4.8 tracks the number of visited states in the Q-learning without planning algorithm. Figures 4.9 and 4.10 tracks the number of visited states in the Q-learning with planning algorithm. It is observed that the number of visited states in the Q-learning without planning has assume a wide range of values between 12 and 27 for Uav_0 and, between 7 and 32 for Uav_1 at the end of the 5 minute simulation. The number of visited states in the Q-learning with planning has assumed a value between 4 and 8 for Uav_0 and, between 4 and 12 for Uav_1 at the end of the 5 minute of simulation. This means that the number of visited states in Q-learning with planning is not only less than the number of visited states in Q-learning without planning but, the range of the number of visited states is also more constricted in the case of Q-learning with planning. This means that the Q-learning without planning explores a wider range of states to identify the optimal policies for the visited states. From 4.12 it is observed that the mean of the number of visited states for Uav_0 in the case of Q-learning is 16.4 and in the case of Q-learning with planning is 6.2. The mean of the number of visited states for Uav_1 in the case of Q-learning is 17.6 and in the case of Q-learning with planning is 7.5.

From figure 4.11 and figure 4.12 it is observed that overall, the Q-learning with planning explores less number of states and still it collects more reward that Q-learning without planning.

CONCLUSION

On observing the trajectory of the UAVs and the convergence of the cumulative reward of UAVs, it can be concluded that both Q-learning and Q-learning with planning are reasonable choices of learning algorithm for the problem of perpetual sparse reward where the rewards reemerge at the same place after a fixed interval of time upon collection.

In comparison to Q-learning learning, the drastic reduction in the number of visited states, the faster convergence of cumulative reward and a higher value of cumulative reward in Q-learning with planning, indicates that Q-learning with planning is a slight improvement in the quality of solution of the learning process. Q-learning with planning is able to collect more cumulative reward but exploring less number of states in comparison to vanilla Q-learning algorithm.

In addition to these quantifiable performances, the planner allows for representing more information about the environment in comparison to vanilla Q-learning. Thus the Q-learning with planning algorithm opens door for more information representation. These information does not only help us improve the performance of the learning algorithm but it also allows for more scope of incorporating multiple behaviours to the agent and multiple goals for the agent.

FUTURE WORK

The primary motivation of this work has been the ability to integrate rich information frameworks such as ontology with learning to use information about the environment to aide in the process of learning.

This work has developed a framework that can be reused to incorporate different characteristic nature of swarms such as- foraging, repulsion and attraction to observe how multi-agent systems are able to learn to emulate swarm characteristics and also observer how multi-agent systems can solve real-world problems by observing swarm behaviours of bees, birds and ants. The ability to incorporate HTN planner with learning will broaden the scope of exploring these swarm characteristics. This is because the swarm characteristics are easy to represent and transfer with modularity. Pyhop is easy to learn and implement. Adding different modules in the HTN planner to emulate these characters in easy.

Since this work is inspired by the capability of creation of HTN planners from ontologies depicted by the work in Freitas *et al.* (2014) the most important future work would be using ontologies for creating the HTN planners for the agents in the environment that are considered as entities in the language of ontology. The HTN planner from ontologies can be explored to track the ease of representing knowledge with HTNs and the performance of the learning algorithm upon using these ontology based HTN planners.

REFERENCES

Adepegba, A. A., S. Miah and D. Spinello, "Multi-agent area coverage control using reinforcement learning", in "The Twenty-Ninth International Flairs Conference", (2016).

Akkaya, I., M. Andrychowicz, M. Chociej, M. Litwin, B. McGrew, A. Petron, A. Paino, M. Plappert, G. Powell, R. Ribas *et al.*, "Solving rubik's cube with a robot hand", arXiv preprint arXiv:1910.07113 (2019).

Alam, T. and L. Bobadilla, "Multi-robot coverage and persistent monitoring in sensing-constrained environments", Robotics **9**, 2, 47 (2020).

Alford, R., U. Kuter and D. S. Nau, "Translating htns to pddl: A small amount of domain knowledge can go a long way.", in "IJCAI", pp. 1629–1634 (2009).

Bechhofer, S., F. Van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, L. A. Stein *et al.*, "Owl web ontology language reference", W3C recommendation **10**, 2, 1–53 (2004).

Chaimowicz, L. and V. Kumar, "Aerial shepherds: Coordination among uavs and swarms of robots", in "Distributed Autonomous Robotic Systems 6", pp. 243–252 (Springer, 2007).

Charlesworth, H. and G. Montana, "Plangan: Model-based planning with sparse rewards and multiple goals", Advances in Neural Information Processing Systems **33**, 8532–8542 (2020).

Erol, K., *Hierarchical task network planning: formalization, analysis, and implementation*, Ph.D. thesis, University of Maryland, College Park (1995).

Faust, A., K. Oslund, O. Ramirez, A. Francis, L. Tapia, M. Fiser and J. Davidson, "Prm-rl: Long-range robotic navigation tasks by combining reinforcement learning and sampling-based planning", in "2018 IEEE International Conference on Robotics and Automation (ICRA)", pp. 5113–5120 (IEEE, 2018).

Freitas, A., D. Schmidt, A. Panisson, F. Meneguzzi, R. Vieira and R. H. Bordini, "Semantic representations of agent plans and planning problem domains", in "International Workshop on Engineering Multi-Agent Systems", pp. 351–366 (Springer, 2014).

Georgievski, I. and M. Aiello, "Htn planning: Overview, comparison, and beyond", Artificial Intelligence **222**, 124–156 (2015).

Ghallab, M., D. Nau and P. Traverso, *Automated Planning: theory and practice* (Elsevier, 2004).

Jaderberg, M., V. Mnih, W. M. Czarnecki, T. Schaul, J. Z. Leibo, D. Silver and K. Kavukcuoglu, "Reinforcement learning with unsupervised auxiliary tasks", arXiv preprint arXiv:1611.05397 (2016).

Kaelbling, L. P., "Learning to achieve goals", in "IJCAI", vol. 2, pp. 1094–8 (Citeseer, 1993).

Kambhampati, S., "A comparative analysis of partial order planning and task reduction planning", ACM SIGART Bulletin **6**, 1, 16–25 (1995).

Koenig, N. and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator", in "2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)", vol. 3, pp. 2149–2154 (IEEE, 2004).

Lekavỳ, M. and P. Návrat, "Expressivity of strips-like and htn-like planning", in "KES International Symposium on Agent and Multi-Agent Systems: Technologies and Applications", pp. 121–130 (Springer, 2007).

Meier, L., D. Honegger and M. Pollefeys, "Px4: A node-based multithreaded open source robotics framework for deeply embedded platforms", in "2015 IEEE international conference on robotics and automation (ICRA)", pp. 6235–6240 (IEEE, 2015).

Michael, N., M. M. Zavlanos, V. Kumar and G. J. Pappas, "Distributed multi-robot task assignment and formation control", in "2008 IEEE International Conference on Robotics and Automation", pp. 128–133 (2008).

Nau, D., "Pyhop kernel description", URL `https://bitbucket.org/dananau/pyhop/src/master/` (2013).

Nau, D. S., S. J. Smith, K. Erol *et al.*, "Control strategies in htn planning: Theory versus practice", in "AAAI/IAAI", pp. 1127–1133 (1998).

Pathak, D., P. Agrawal, A. A. Efros and T. Darrell, "Curiosity-driven exploration by self-supervised prediction", in "International conference on machine learning", pp. 2778–2787 (PMLR, 2017).

Quigley, M., K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng *et al.*, "Ros: an open-source robot operating system", in "ICRA workshop on open source software", vol. 3, p. 5 (Kobe, Japan, 2009).

Savinov, N., A. Dosovitskiy and V. Koltun, "Semi-parametric topological memory for navigation", arXiv preprint arXiv:1803.00653 (2018).

Schmittle, M., A. Lukina, L. Vacek, J. Das, C. P. Buskirk, S. Rees, J. Sztipanovits, R. Grosu and V. Kumar, "Openuav: A uav testbed for the cps and robotics community", in "2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)", pp. 130–139 (2018).

Sutton, R. S. and A. G. Barto, "A temporal-difference model of classical conditioning", in "Proceedings of the ninth annual conference of the cognitive science society", pp. 355–378 (Seattle, WA, 1987).

Sutton, R. S. and A. G. Barto, *Reinforcement learning: An introduction* (MIT press, 2018).

Tesauro, G. *et al.*, "Temporal difference learning and td-gammon", Communications of the ACM **38**, 3, 58–68 (1995).

Weld, D. S., "An introduction to least commitment planning", AI magazine **15**, 4, 27–27 (1994).

Zeng, S., Y. Zhu and C. Qi, "Htn-based multi-robot path planning", in "2016 Chinese Control and Decision Conference (CCDC)", pp. 4719–4723 (IEEE, 2016).

Zhang, A., S. Sukhbaatar, A. Lerer, A. Szlam and R. Fergus, "Composable planning with attributes", in "International Conference on Machine Learning", pp. 5842–5851 (PMLR, 2018).