IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 24 March 2024, accepted 9 April 2024, date of publication 18 April 2024, date of current version 26 April 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3390796

RESEARCH ARTICLE

Incorporating Online Learning Into MCTS-Based Intention Progression

CHENGCHENG SONG[1], **YUAN YAO**[2], **(Member, IEEE), AND SIXIAN CHAN**[1]
[1]School of Computer Science and Technology, Zhejiang University of Technology, Hangzhou 310023, China
[2]School of Computer Science, University of Nottingham Ningbo China, Ningbo 315100, China

Corresponding author: Yuan Yao (Yuan.Yao@nottingham.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61906168, in part by Zhejiang Provincial Natural Science Foundation of China under Grant LY23F020023, and in part by the Yongjiang Talent Introduction Program under Grant 2022A-234-G.

ABSTRACT Agents have been applied to a wide variety of fields, including power systems and spacecraft. Belief-Desire-Intention (BDI) agents, as one of the most widely used and researched architectures, have the advantage of being able to pursue multiple goals in parallel. The problem of deciding "what to do" next at each of the agent's deliberation cycle is therefore critical for BDI agents, which is defined as the intention progression problem (IPP). Among all existing approaches to IPP, the majority of approaches have overlooked the significance of runtime historical data, thereby limiting the adaptability and decision-making capabilities of agents. In this paper, we propose to incorporate online learning into the current state-of-the-art intention progression approach S_A to overcome the above limitations. This approach not only prevents S_A from consuming computational resources on ineffective and inefficient simulations, but also significantly improves the execution efficiency of the agent. Especially when dealing with large-scale problem domains, this improvement significantly enhances the planning capability of the agents. In particular, we have proposed the SA_Q and SA_L schedulers, both of which can learn how to generate "reasonable" rollouts during the simulation phase of MCTS based on historical simulation data at run time. We compare the performance of our approach with the state-of-the-art S_A in a range of scenarios of increasing difficulty. The results demonstrate that our approaches outperform S_A, both in terms of the number of goals achieved and the computational overhead required.

INDEX TERMS BDI agents, intention progression problem, Monte-Carlo tree search, online learning.

I. INTRODUCTION

The Belief-Desire-Intention (BDI) model [1], which originated from philosophical work on practical reasoning [2], is one of the most popular architectures for implementing agent-based systems, especially in complex environments [3]. In the BDI model, the behaviour of an agent is specified by its mental state which consists of beliefs, goals and intentions. Beliefs represent the agent's information about the environment (and itself). Desires (or goals) indicate the state of the environment that the agent wants to bring about. Plans are the means by which an agent can achieve its goals. Each plan contains a sequence of execution steps which are either primitive actions that can directly change the state of the environment or subgoals that are in turn achieved by sub-plans. Once a plan is selected to achieve a particular goal, an intention is formed. A key feature of BDI agents is their ability to simultaneously pursue multiple goals in parallel. To do so, a BDI agent needs to iteratively select and decide which of its current intentions should be progressed next, and if the next step in the selected intention is a (sub)goal, the agent also needs to decide which plan should be used to achieve the goal. Each iteration in the repetitive process is called the agent's deliberation cycle, and the problem of deciding "what to do next" is termed the *Intention Progression Problem (IPP)* [4].

A number of approaches have been proposed in the literature to address the intention progression problem

The associate editor coordinating the review of this manuscript and approving it for publication was Jianxiang Xi.

from different aspects. Thangarajah et al. [5], [6], [7], [8] proposed an approach based on summary information (SI) that avoids conflicts between intentions by reasoning about the necessary and possible conditions to achieve a top-level goal. Waters et al. [9], [10] proposed a coverage-based (CB) approach that selects the intention with the lowest coverage for execution, i.e., the intention with the highest probability of becoming non-executable due to environmental changes. These approaches mostly rely on pre-computed information to guide the decision-making process of agents, to some extent overlooking the significant value embedded in historical data. The most promising approach to IPP to date is perhaps the S_A scheduler proposed by Yao et al. [11], [12], which is based on the Monte-Carlo Tree Search (MCTS) [13], [14]. The S_A scheduler uses pseudorandom simulations to guide the expansion of the search tree and returns the "best" action to be performed in each deliberation cycle, where the agent's intentions are denoted as goal-plan trees as in [15].

Although Yao et al. have shown their approach outperforms SI and CB in both static and dynamic environments [11], it also fails to fully leverage the runtime historical data in its decision-making process. Firstly, the simulation policy used in the S_A scheduler during its simulation phase is totally random. While this policy allows the scheduler to explore all possible moves during the simulation process, its randomness also means that it may waste computational resources on unnecessary or inefficient simulations. This lack of "preferences" simulation will significantly affect the effectiveness and the efficiency of S_A. Secondly, the S_A scheduler re-generates the MCTS search tree at the beginning of each deliberation cycle [11], none of the existing statistics or the MCTS search tree is considered in future steps. Therefore, we need to address two aspects of how to learn the historical simulation data and how to use the historical simulation data generated in the previous deliberation cycles to assist the intentions scheduling in later steps.

In this paper, we introduce a novel intention scheduling algorithm by incorporating the online learning mechanism into the S_A scheduler. Specifically, our approach involves continuously accumulating and learning from historical simulation experiences through online learning, i.e., the agents reduce resource wastage on "bad" simulations through online learning, and guide the agents to prioritize actions that have performed well in historical simulations in future simulations, thus greatly improving the effectiveness and efficiency of decision-making. Therefore, our approach maintains the original advantages of the S_A scheduler while achieving the improvement and optimization of the traditional approach by incorporating an online learning mechanism.

In this paper, we first introduce two approaches that can be used as the simulation policy for MCTS. The first approach is based on the traditional Q-learning and the second one is based on a hierarchical tree structure called *State-Action* tree. Both approaches take the simulation results generated in previous deliberation cycles into account and then generate probability distributions based on the statistics. We then propose two variations of S_A scheduler called SA_Q and SA_L. Unlike S_A which generates totally random rollouts in its simulation phase, SA_Q and SA_L use the proposed simulation policies to generate "intelligent" simulations at run time. Finally, we evaluate the performance of SA_Q and SA_L, and compare them to the state-of-the-art S_A scheduler in both static and dynamic environments. The preliminary experimental results suggest that both SA_Q and SA_L can achieve the same number of goals as S_A with less time, and if the computational time is fixed, SA_Q and SA_L can achieve more goals compared to S_A scheduler. Moreover, in almost all situations SA_L performs slightly better than SA_Q.

The rest of the paper is organized as follows. In Section II, we provide a brief overview of the terms related to BDI agents and formally define the intention progression problem. A brief introduction to the S_A scheduler together with a short discussion on how to improve its performance is included in Section III. In Section IV, we present two different approaches to learning how to generate rollouts based on previous simulation results, namely the Q-learning-based approach and the *State-Action* tree approach. The proposed approaches are then incorporated with the MCTS-based scheduling in Section V to form the SA_Q and SA_L scheduler. We then evaluate the performance of the proposed schedulers in Section VI, and finally, a brief discussion on the related works and future research directions is provided.

II. PRELIMINARIES

We start this section by giving the definition of beliefs, goals and plans, and then briefly introduce the goal-plan tree structure which can be used to represent agents' intentions. Finally, a formal definition of the intention progression problem based on the goal-plan tree structure is presented.

A. BELIEFS, GOALS, PLANS AND ACTIONS

1) BELIEFS

The agent's beliefs represent its information about the environment, i.e., what the agent believes to be true. For simplicity, we assume that the agent's beliefs B is a finite set of literals (proposition p or its negation $\neg p$):

$$B = \{b_1, \ldots, b_n\} \tag{1}$$

B is updated at each cycle based on the agent's perceptions, i.e., the agent updates its beliefs after sensing the environment at each cycle. We further assume that the beliefs in B are consistent, i.e., there is no p such that both $p \in B$ and $\neg p \in B$ are true.

2) GOALS

The goals delegated to BDI agents can be divided into different categories based on their properties and how they will be pursued. One commonly used taxonomy is to classify goals as achievement goals and maintenance goals. Achievement goals specify the state of the environment the agent is trying to bring about, while maintenance goals require the agent to maintain a particular state of the environment for a while. In this paper, we focus on achievement goals. Each goal G is associated with a set of plans P_1,…,P_n that achieve G. Once an achievement goal is achieved, the goal itself is dropped by the agent.

3) PLANS

Plans are the means by which an agent can achieve its goals. For each top-level goal, the agent may have multiple but at least one plan to achieve it. Each plan p_i has its context conditions $\chi = pre(p_i)$ which is a set of literals specifying the conditions that must be met for the plan to be applicable. Once a plan is selected for execution, all the execution steps within this plan will be executed. Each execution step is either a primitive action that can directly change the state of the environment or a sub-goal that is in turn achieved by a subplan.

4) ACTIONS

An agent can perform a set of primitive actions in the environment.

$$A = \{a_1, \ldots, a_k\} \tag{2}$$

Each action has its corresponding pre- and post-conditions. Pre-conditions specify the states of the environment which must hold for the action to start execution. Post-conditions are the state of the environment brought about by executing the action. The pre- and post-conditions of an action $a_i(1 \leq i \leq k)$ are represented as a set of literals $\phi = pre(a_i)$ and $\varphi = post(a_i)$. If the pre-conditions of an action hold, then the post-conditions of the action will be achieved after executing the action.

B. GOAL-PLAN TREE

The Goal-Plan Tree (GPT) is a hierarchical tree structure that can be used to represent the relationships between goals and plans [5], [8]. The root of a goal-plan tree is a top-level goal node which contains the name of the goal, and the (goal) conditions that need to be achieved. Its children are "OR" nodes, each of which is a possible plan to achieve the goal, i.e., only one of the plans will be selected for execution to achieve the goal. Each plan node contains information about the name and the context condition of the plan. Once a plan node is selected for execution, all its child nodes representing the subgoals in that plan need to be achieved. In order to facilitate the deliberation of intention progression at the action level, we use the version of GPTs defined in [15], where each child of a plan node can be either a subgoal node or an action node. All action nodes in the GPT are leaf nodes that only contain the pre- and post-condition of the specified actions. In order to achieve a plan, all the subgoals in the plan should be achieved and all actions in the plan should be successfully executed in a pre-defined order.

Fig. 1 shows an example of a goal-plan tree, straight lines connect goal nodes to the plan nodes that are designed to achieve the goal, e.g., P_0 and P_1 are two plan nodes to achieve the goal node g_0; dashed arrows connect plan nodes to their respective execution steps, e.g., g_1, a_0, a_1 are the execution steps of plan node P_0. The steps in the plan should be executed in a pre-determined order as represented by the "Arrow", e.g., the execution steps in P_1 need to be executed in the following order: a_3, a_4 and then a_5.

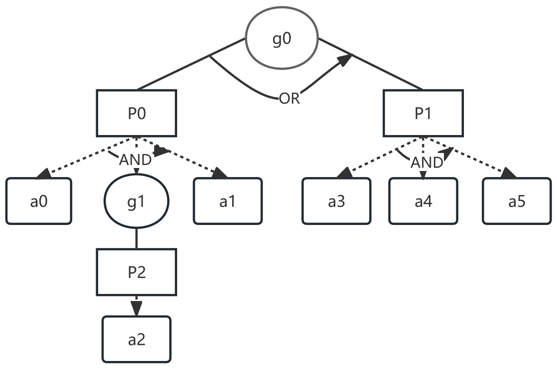

FIGURE 1. An example of GPT.

C. INTENTION PROGRESSION PROBLEM

As discussed, a key problem for BDI agents is to decide how to progress their intentions at each of its deliberation cycles. This problem is formally defined as the *Intention Progression Problem (IPP)* [4], [16] which consists of two sub-problems, i.e., the intention selection problem and the plan selection problem. In an intention selection problem, a BDI agent needs to decide which intention should be executed next. The execution order of the agent's intention could decide the order in which the agent's goals are achieved, and potentially cause conflicts while achieving the goals, e.g., certain steps or intentions need to be finished before others. Once an intention is selected, the agent also needs to decide which plans should be used if the next step in the selected intention is a subgoal, which is termed the plan selection problem. As with intention selection, the results of plan selection could affect how agents' goals are achieved and if they can be achieved simultaneously.

In this paper, we define the intention progression problem as a tuple $< B, I, f_u >$, where B is the agent's current belief base, I is the agent's current intentions, and f_u is a utility function specifying how the intentions should be progressed, e.g., f_u might be designed to favour solutions that can achieve more goals. Formally, the intentions of an agent are defined as a set of pairs $I = \{(t_1, s_1), \ldots, (t_n, s_n)\}$, where t_i are the goal-plan trees representing all possible ways of achieving the agent's top-level goals, s_i are their corresponding current step pointers which point to either primitive actions or subgoals. Initially, the current step pointer of each goal-plan tree t_i is set to its corresponding top-level goal g_i. Suppose, we use $next(s_i)$ to represent the step following the current step s_i of t_i. If s_i is a sub-goal, advancing the current step requires selecting a plan for the sub-goal and setting s_i to be the first step (action or subgoal) of the selected plan. If s_i is an action, then $next(s_i)$ is the step that follows s_i in the same plan. Note that if s_i is the last step in a plan, $next(s_i)$ is the next step in the parent plan of the current plan. Together, we could use t_i and s_i to define what remains to be done in order for the agent to achieve its goals. The progression of an intention to achieve a top-level goal g_i involves traversing the goal-plan tree t_i to obtain a path that specifies a series of sequentially executed plans, actions, subgoals and sub-plans. If executed successfully, g_i is achieved. The intention progression of multiple intentions is equivalent to the interleaving of multiple paths. The solution to the intention progression problem is therefore a policy Π that selects an action to execute at each deliberation cycle which maximizes the given utility function f_u.

III. MCTS-BASED INTENTION PROGRESSION

In this section, we briefly introduce the Monte-Carlo-Tree-Search-Based intention progression scheduler, S_A, and then discuss how the original S_A could be further improved with historical data while maintaining its generality.

A. S_A SCHEDULER

The S_A scheduler proposed by Yao and Logan [11] was developed based on the well-known Monte-Carlo Tree Search (MCTS) which has shown its potential in solving complex combinatorial problems [13], [17] in recent years. The scheduling procedure is a repeated process of building a search tree based on pseudo-random simulations. Once the pre-defined computational budget is reached, e.g., time, memory or number of iterations, the procedure will halt and the "best" next step will be selected and returned. In the MCTS search tree, each node n represents a possible state in the problem domain. We use $state(n)$ to refer to the state in node n. In the intention progression problem, a state is defined as a tuple $<B, I>$, where B and I are the agent's current belief base and intentions respectively. Edges that connect a parent node to a child node in the search tree are "steps", by executing which the state in the parent node will be transited to the state in the corresponding child node. In S_A, "steps" are primitive actions that can directly change the state of the environment.[1] Starting from the root node which represents the agent's current state, S_A iteratively builds a search tree. Each iteration consists of 4 phases: selection, expansion, simulation and back-propagation (see Algorithm 1).

1) SELECTION

In the selection phase, a *tree policy* is applied to the tree to select the most "urgent" leaf node for expansion. A good *tree policy* needs to balance exploration (traversing nodes that have rarely been visited) and exploitation (favouring steps that previously led to strong returns). In S_A, a modified version of Upper Confidence bounds applied to Trees (UCT) [18] is used as the *tree policy*, which models the selection of child nodes as the k-armed bandit problem [19]. Starting from the root node n_0, S_A recursively selects the child node with the highest UCT value until a leaf node n_e is reached (lines 4-7).

2) EXPANSION

The selected node n_e is then expanded by adding new child nodes to it (line 8). In the case of S_A scheduler, child nodes of n_e represent all the states that are reachable from $state(n_e)$. A state $state(n_r)$ is considered to be reachable from $state(n_e)$ if it can be reached by executing a primitive action a_r from $state(n_e)$. We further require that a_r is one of the possible next steps of the agent's intentions in $state(n_e)$, and a_r is currently executable, i.e., the precondition of a_r holds in $state(n_e)$.

3) SIMULATION

One of the newly generated nodes n_s is then randomly selected for simulation (line 9). The rollouts of n_s are generated according to the *simulation policy* (line 10) and the simulation results are evaluated based on the given unity function f_u (line 10). The S_A scheduler uses a random *simulation policy* to produce the possible rollouts, i.e., the scheduler randomly selects and executes available next steps until all goals have been achieved or no intention can be further progressed.

4) BACK-PROPAGATION

Finally, the simulation results are back-propagated to all the nodes on the path from node n_s to the root node n_0 (line 11).

Once the pre-defined computational budget is reached (and the algorithm halts), the edge to the "best" child node of n_0 will be returned as the primitive action to be executed next (line 13). The most commonly used functions f_{best} include those that select a node with the highest simulation result, highest average, etc.

Algorithm 1 Basic MCTS Procedure

```
1:  function MCTS($B, I, f_u, f_{best}$)
2:      Create a root node $n_0$ with state $s_0 =< B, I >$
3:      while within computational budget do
4:          $n_e \leftarrow n_0$
5:          while $n_e$.hasChild() do
6:              $n_e \leftarrow$ TreePolicy($n_e$)
7:          end while
8:          Expand($n_e$)
9:          $n_s \leftarrow n_e$.getRandomChild()
10:         $\Delta \leftarrow$ SimulationPolicy($n_s, f_u$)
11:         BP($n_s$, $\Delta$)
12:     end while
13:     return $edge(n_0, f_{best}(n_0))$
14: end function
```

[1]Note that executing a primitive action may also involve the process of plan selection.

B. S_A ENHANCEMENT

Although S_A has been proven to outperform most of the existing intention progression algorithms, there are still some issues that remain unsolved. For example, S_A may not be very efficient for problems with sufficiently large problem domains, as the scheduler is likely to waste time on some obviously "bad" simulations. Additionally, the current S_A scheduler lacks the ability to make use of "historical data" and "domain knowledge" to improve its performance in specified problem domains.

There are several ways the current S_A could be improved including the changes to the *tree policy*, the *simulation policy* or the *expansion* function. The *tree policy* in MCTS will directly affect the node to be investigated/expanded next, and thus determine the direction of the tree expansion. In a tree search, it is critical to ensure the search tree is expanding in the right direction. The *expansion* function, on the other hand, decides the nodes to be added to the tree. By pruning the search tree or adding constraints to the node expansion, we could ignore the nodes led by obviously "bad" steps. Finally, the *simulation policy* could be adjusted to generate more "promising" rollouts, avoid obviously "bad" choices during the simulation, and thus improve the reliability of the simulation results. In this paper, we focus on improving the current simulation policy to enhance the performance of S_A. Although both the changes to the *tree policy* and the *expansion* function might be useful to improve the performance of S_A, they are not within the scope of this paper.

IV. LEARNING-BASED SIMULATIONS

As discussed, one possible way of enhancing the performance of S_A is to optimise the current "simulation policy". Instead of running totally random simulations as in S_A, it might be beneficial to properly bias the choice of actions during the simulation phase. That is, the scheduler is more favouring the selections that are likely to produce good rollouts and tries to avoid selections that lead to failure or bad performance. For example, we all know that it will cause a problem if we try to leave the lecture room while attending a lecture, thus such choice needs to be avoided in the simulation. The basic idea of optimising the current "simulation policy" is to learn from previous simulations what are good and bad simulation choices, and then bias the choice towards good selection during the simulation phase. Initially, it is difficult to judge if a selection during the simulation is good or bad without giving any domain knowledge. However, as the scheduler keeps running simulations, the performance of the simulation policy will be significantly improved. In this section, we introduce two ways of "learning" biasing simulation choices from historical data, namely the Q-learning based approach and the *State-Action* tree approach. Both of them work as online learning mechanisms (i.e., learning at runtime), and are able to return the most promising choices based on the results of previous simulations.

A. Q-LEARNING BASED APPROACH

Q-learning [20] is a popular reinforcement learning algorithm that has been widely used in games and multi-agent systems [21], [22]. It relies on a two-dimensional Q-table to store the Q-value for each possible state-action pair and optimizes the policy by continuously updating the Q-value. During the learning process, the agent interacts with the environment to update the Q-value and selects the next action based on the updated Q-value. Notably, Q-learning tends to exploit existing knowledge by selecting the action with the maximum Q-value for the current state. This aligns well with the idea of combining online learning to bias the simulation selection in favor of more promising actions. The following are brief descriptions of how Q-table can be used to represent historical simulation results, and when and how the Q-value will be updated.

1) $Q(S_T, A_T)$: ACTION-VALUE FUNCTION

The Q-value stored in the Q-table can be accessed through the Q-function which takes a state-action pair (s_t, a_t) as its input. In the intention progression problem, the state s_t is a pair of the agent's current beliefs and intentions (same as the $state(n)$ in S_A), the action a_t represents a primitive action to be executed in s_t. During the simulation phase, the scheduler could simply retrieve the Q-value of executing action a_t in state s_t by calling $Q(s_t, a_t)$.

2) Q-TABLE UPDATE

Once the reward of a simulation (Δ) is generated by a utility function f_u, the Q-values for all the state-action pairs that appear in the simulation will be updated based on the following formula:

$$Q(s_t, a_t) = Q(s_t, a_t) + \omega[\Delta + \gamma \cdot MaxQ(s_r, a_r) - Q(s_t, a_t)] \tag{3}$$

where s_t and a_t denote the state-action pair to be updated, s_r denotes the state resulting from executing action a_t in state s_t. a_r represents a possible action to execute in state s_r. $MaxQ(s_r, a_r)$ is therefore the maximum Q-value for executing an action in state s_r, and finally Δ is the reward value. The learning rate is denoted as ω. It determines the extent to which the new Q-value contributes to the overall Q-value during each update. The discount factor γ determines the degree to which current rewards influence future rewards. If the discount factor is set too high, the algorithm will prioritise future rewards. Conversely, if the discount factor is set too low, it will prioritise immediate rewards.

Traditional Q-learning algorithm initializes the Q-table with all possible states and actions, where each cell stores the expected return value (Q-value) of performing a certain action in a specific state. The agent gradually learns to select optimal actions in different states based on the Q-value. However, this method is only suitable for domains with a small state space. The state space in IPP is significantly large due to the fact that states are represented by not only the agent's internal intention stacks but also its current beliefs. Therefore, certain modifications need to be adjusted to the traditional Q-learning algorithm. Specifically, we limit the state space for learning by setting the state filtering criteria, which greatly reduces the number of possible state-action pairs and improves the efficiency of the algorithm. The specific details on how to set these criteria will be discussed in detail in later sections.

B. STATE-ACTION TREE

Instead of using the Q-table to store all possible state-action pairs, we propose to use a tree structure to store historical data named *State-Action* tree. Here, we provide a formal definition of the *State-Action* tree structure together with a description of how the historical simulation results can be stored and updated based on this structure.

1) DEFINITION

Each node N_i in the *State-Action* tree is represented as a tuple $< s_i, a_i, T_i, V_i >$, where

- s_i indicates the current state of the agent, i.e., a tuple of the agent's current beliefs and intentions < B, I > as in the MCTS tree.
- a_i is a primitive action that can be executed in state s_{i-1}, where s_{i-1} is the state in the parent node of N_i, N_{i-1}. Executing action a_i in state s_{i-1} will transmit the agent's state from s_{i-1} to s_i.
- T_i represents the number of times action a_i has been chosen for execution in state s_{i-1} in previous simulations.
- V_i represents the total simulation value of N_i, i.e., the sum of the reward for the simulations that execute action a_i in state s_{i-1}.

The first two parameters together represent the agent's current state and the decision made in its preceding state.[2] How good the decision is can be estimated through the associated simulation statistics (i.e., the value of T_i and V_i). Unlike the leaf nodes in MCTS which can be potentially further expanded, all leaf nodes in the *State-Action* tree are terminal states representing the end of a simulation. A path from the root node to a leaf node, therefore, can be seen as a possible rollout starting from the root node.

2) TREE GENERATION AND UPDATE

When generating a new *State-Action* tree, a root node representing the current state of the agent will be created, i.e., $N_0 = < s_0, \delta, 0, 0 >$, where s_0 is the agent's current state, δ means there is no primitive action executed prior to state s_0,[3] and the statistics for this node are all initialised to 0. We then expand the tree and update its statistics based on the given simulation paths. A simulation path from a node n_i until a terminal state is represented as follows:

$$P = [(s_i, a_i), (s_{i+1}, a_{i+1}), \ldots, (s_t, a_t)]$$

where $s_i = state(n_i)$, a_i is an edge that connects n_i to its parent node,[4] s_t is a terminate state, and executing action a_{k+1} in state s_k will result in a new state s_{k+1} for all $i \leq k \leq t-1$. So, the definition of state-action pairs in a path is essentially the same as the state and action in a *State-Action* tree node. Given a simulation path P together with its associated reward Δ, the *State-Action* tree will be expanded and updated as in Algorithm 2.

Algorithm 2 Update the *State-Action* tree Based on a Rollout

```
1:  function Update($N_0$, $P$, $\Delta$)
2:      $N_c \leftarrow N_0$
3:      for each $(a_j, s_j) inP$ do
4:          $N_x \leftarrow SearchFor(s_j, N_c)$
5:          if $N_x$ == NULL then
6:              $N_x \leftarrow < s_j, a_j, 0, 0 >$
7:              $N_c.AddChild(N_x)$
8:          end if
9:          $N_c \leftarrow N_x$
10:         $N_c.UpdateValue(\Delta)$
11:     end for
12: end function
```

Starting from the root node N_0, Algorithm 2 iteratively searches for the nodes corresponding to each state-action pair in the given simulation path, and updates their statistics based on the reward value Δ. More precisely, the *Update* function in Algorithm 2 does not only update the statistics for the existing nodes in the *State-Action* tree but also expands the tree if the nodes specified by the state-action pairs haven't been included in the tree yet, i.e., the *SearchFor* function fails (see Line 5-7).

Compared to the Q-learning based approach, the *State-Action* tree offers several advantages. Firstly, the relationship between different state-action pairs naturally fits in the tree structure. Searching for a node corresponding to a particular state-action pair in a tree structure is more efficient compared to finding a state in the Q-table. Secondly, using T_i and V_i in the tree structure can better reflect the stability and reliability of state-action pairs compared to Q-value. This is because the number of visits reflects the level of exploration of the state-action pair, while the total simulation value reflects the cumulative evaluation of the state-action pair. Furthermore, when it comes to irrelevant state cleanup, pruning the *State-Action* tree is sufficient, whereas the Q-table requires sequential traversal of each state. Therefore, pruning in a tree structure is more effective and time-saving compared to clearing the Q-table. However, the update process of *State-Action* tree is more complex compared to the Q-table, as it involves the process of expanding the current search tree.

[2]Note that the state-action pairs here are different from those defined in the Q-table.

[3]There may be actions executed before s_0 in the entire scenario, but they are not considered in the *State-Action* tree.

[4]$a_i = \delta$ if n_i is the root node of the MCTS tree.

V. INTENTION SCHEDULING WITH ONLINE LEARNING

In this section, we present two novel simulation policies based on Q-table and *State-Action* tree. For ease of reference, we use SA_Q to refer to the scheduler which uses Q-table in its simulation policy, and SA_L for the scheduler which generates simulation rollouts based on the *State-Action* tree.

A. SA_Q SCHEDULER

Compared to the S_A scheduler, SA_Q has significantly modified its *SimulationPolicy* function (see Algorithm 3).

The new simulation policy requires five parameters as its input: an MCTS node n_s, a Q-table QT, and three constant values ε, Γ and ν. The value of ε is a floating number in the range of [0, 1], which controls the balance between selecting good actions based on historical data and exploration of other potentially good actions. Γ and ν are the total simulation results and the total number of simulations from all previous runs.

Starting from line 2, the proposed simulation policy first created a list of state-action pairs (*List*) from the given node n_s all the way to the root node n_0, representing the actions taken and the state achieved prior to the state in n_s. A major difference here is that each pair in the *List* is defined as (s_j, a_{j+1}) instead of (s_j, a_j) as in Section IV-B, where s_{j+1} is the subsequent state of executing action a_{j+1} in state s_j. Given the current state s_i, we can generate the set of executable actions As in s_i based on the agent's current beliefs and intentions. The simulation policy then iteratively selects and executes actions until a terminal state is reached (line 5), i.e., either all goals have been achieved or no actions can be executed from s_i. We also record all the state-action pairs that appear in this iterative procedure and add them to *List* in order to update the Q-table after the simulation. Remember that the definition of state-action pair here differs from that of Section IV-B. Consequently, we need to add the state-action pair to *List* before executing the selected action (lines 11-12). Once a terminal state is reached, the overall performance of this simulation is assessed by calling the unity function f_u (line 15).

Similar to the Q-learning [20], we use the ε-*greedy* policy to iteratively select actions during the simulation process (line 6-10). That is, in each iteration, the policy has a 1 - ε probability to select the action with the largest Q-value, and a probability ε to randomly select an executable action (which is the same as the policy in S_A). One exception would be when there is no such $Q(s_i, A')$ in the current Q-table, i.e., no action selection has been learnt for state s_i or all the action selections in previous simulations are considered as "bad" attempts,[5] the policy will make a random selection rather than selecting the "best" action. This is especially useful when the agent has no historical data, i.e., the Q-table is empty, and ensures our approaches will still work even without any previous simulations. The reason to use the ε-*greedy* policy in the action selection procedure is to ensure the agent has the opportunity to select non-optimal actions and thus to avoid being stuck with local maximal.

[5]This will happen as we do not include Q-values for all possible states at the beginning, instead, we generate states and Q-values along with the simulation.

Algorithm 3 Simulation Policy With Q-table

```
1:  function SimulationPolicy($n_s$, $QT$, $\varepsilon$, $\Gamma$, $\nu$)
2:      $List \leftarrow getStateActions(n_s)$
3:      $s_i \leftarrow state(n_s)$
4:      $As \leftarrow nextActions(s_i)$
5:      while $As$ is not Empty do
6:          if Random() $\le \varepsilon$ then
7:              $A \leftarrow$ RandomSelect($As$)
8:          else if BestAction($QT$, $s_i$) $\neq$ NULL then
9:              $A \leftarrow$ BestAction($QT$, $s_i$)
10:         end if
11:         $List.add((s_i, A))$
12:         $s_i \leftarrow Execute(s_i, A)$
13:         $As \leftarrow nextActions(s_i)$
14:     end while
15:     $\Delta \leftarrow f_u(s_i)$
16:     $\Gamma \leftarrow \Gamma + \Delta$
17:     $\nu \leftarrow \nu + 1$
18:     if $\Delta \ge \frac{\Gamma}{\nu}$ then
19:         $QT$.update($List$, $\Delta$)
20:     end if
21: end function
```

To address the problem of an excessively large state space raised in the previous section, we set the criterion that each simulation is considered worth learning only when its reward is greater than the current average reward for all previous simulations, i.e., $\frac{\Gamma}{\nu}$. At the end of each simulation, the function f_u is used to evaluate the performance of the simulation (denoted as Δ). If the value of Δ is higher than $\frac{\Gamma}{\nu}$, then the policy will update QT based on the list of state-action pairs in *List* and the reward Δ (lines 18-19). As we have included all the state-action pairs executed or achieved before the simulation and during the simulation, we can therefore represent the *List* as $\{(s_0, a_1), \ldots, (s_i, a_{i+1}), \ldots, (s_{t-1}, a_t)\}$, where s_0 is the state in the root node and s_t is the terminal state in the simulation. The function QT.update($List$, Δ) will update each $Q(s_i, a_{i+1}) \in List$ based on Equation 3.

In a static environment, after reaching a given computational budget, the agent returns the best child node of the MCTS search tree's root node and executes the corresponding action. Since the environment remains unchanged, the Q-table updated in previous deliberations can be utilized in subsequent deliberations. Therefore, maintaining the current Q-table unchanged, the Q-table guides the simulation in subsequent deliberations and continues to be updated. One possible optimization here is to remove all Q-functions that will no longer be used for future selections. These Q-functions include those that are not selected (and will never be selected) for execution and their corresponding subsequent state-action pairs that appear in previous simulations. In the case of a dynamic environment, the environment itself may change in addition to the changes made by the agent. As a result, potentially more Q-value will be affected. In the cases where most of the previously learned Q-table are no longer applicable, the agent will initialise the Q-table and relearn Q-value based on the new simulation rollouts. Otherwise, the approach is similar to that of a static environment, where the Q-table remains unchanged and is used to guide simulations in the next deliberation and continues to be updated. This enables the agent to adapt to constantly changing environments and learn new optimal strategies.

B. SA_L SCHEDULER

Similar to SA_Q, the SA_L scheduler has significantly changed its simulation policy by incorporating a *State-Action* tree to guide the selection of actions during the simulation phase. Algorithm 4 shows the simulation policy based on the *State-Action* tree.

Algorithm 4 Simulation Policy With *State-Action* tree

```
1:  function SimulationPolicy($n_s$, $N_0$, $\varepsilon$, $\Gamma$, $\nu$)
2:      $List \leftarrow getStateActions(n_s)$
3:      $s_i \leftarrow state(n_s)$
4:      $As \leftarrow nextActions(s_i)$
5:      while $As$ is not Empty do
6:          $N_i \leftarrow SearchFor(s_i, N_0)$
7:          if Random() $\leq \varepsilon$ then
8:              $A \leftarrow$ RandomSelect($As$)
9:          else if $N_i \neq$ NULL or $N_i.getChild \neq$ NULL then
10:             $A \leftarrow$ BestChild($N_i$)
11:         end if
12:         $s_i \leftarrow Execute(s_i, A)$
13:         $List.add((s_i, A))$
14:         $As \leftarrow nextActions(s_i)$
15:     end while
16:     $\Delta \leftarrow f_u(s_i)$
17:     $\Gamma \leftarrow \Gamma + \Delta$
18:     $\nu \leftarrow \nu + 1$
19:     if $\Delta \geq \frac{\Gamma}{\nu}$ then
20:         Update($N_0$, $List$, $\Delta$)
21:     end if
22: end function
```

As with the Q-table based simulation policy, the proposed simulation policy for SA_L scheduler also requires five parameters as inputs, which are essentially the same as those used in Algorithm 3, except that Algorithm 4 requires the root node of a *State-Action* tree as its input rather than a Q-table. Similar to Algorithm 3, the proposed simulation policy first creates a list of state-action pairs (*List*) to record the states and actions that were achieved or executed prior to the state in n_s. Each state-action pair $(s_i, a_i) \in List$ indicates that executing action a_i in the state s_{i-1} leads to a subsequent state s_i. Assuming $n_0, \ldots, n_s$ is a path from the root node of the MCTS tree to the selected node n_s, and we use $s_0, \ldots, s_n$ to represent the state in each of these nodes, then the list of state-action pairs in *List* is defined as $(s_0, a_0), \ldots, (s_i, a_i), \ldots, (s_n, a_n)$. Note that, a_0 here can be empty if we don't know which action leads to the state of the root node (we will discuss it more later in this section). Similarly, a set of executable actions in state s_i (As) are generated based on the agent's current belief base and intention base. We then iteratively select and execute actions based on the given *State-Action* tree until a terminal state is reached (lines 5-15). During the simulation process, all the state-action pairs that appear in the rollouts will be recorded and added to *List*. Here, we need to execute the selected action before adding the state-action pair to *List* in order to get the state that results from executing action A.

In the iterative action selection procedure, we use a similar ε-*greedy* policy to balance between the selection of known "best" action and the exploration of all other actions. Same as in Algorithm 2, the function *SearchFor* is used to find the tree node corresponding to state s_i from the root node N_0. If there is no such tree node or there is no further action selection for state s_i in the *State-Action* tree, then we randomly select an executable action. Otherwise, the policy will have a 1 - ε probability of selecting the "best" action indicated by the *State-Action* tree. Due to the influence of the total number of visits, the node with the highest average simulation value is often the one that has been simulated and visited more frequently. The node with the highest average reward may provide more reliable and accurate information and is likely to contain the optimal solution. Selecting the node with the highest average reward value can help us converge to the optimal solution faster. We therefore consider an action A with an average reward R to be optimal in state s_i if there is no child node $(s_{i+1}, a_{i+1}, T_{i+1}, V_{i+1})$ of N_i in the given *State-Action* tree that has a larger average reward than R, i.e., $\frac{T_j}{V_j} > R$. By focusing on nodes with higher average simulation values, we can more effectively utilize prior knowledge and speed up the search process. At the end of each simulation, the reward value Δ which is larger than the overall average reward will be used to update the *State-Action* tree as in Algorithm 2.

Similar to the Q-table, the *State-Action* tree also needs to clean up irrelevant states after executing the optimal action returned by MCTS. However, the difference is that cleaning up irrelevant state in the *State-Action* tree only requires pruning out the branches caused by other actions performed in the state before the action is executed. At the same time, the root node of the *State-Action* tree is also updated, i.e., the root node is updated to the tree node corresponding to the state after the agent executes the optimal action. In the dynamic environment, the changes made by the environment itself may significantly affect future simulations. In cases where the current state (i.e., the state after executing the "best" action in the previous cycle) cannot be found in the *State-Action* tree, the agent needs to initialize the *State-Action* tree and relearn the simulation policy based on new simulations. Otherwise, we could simply update the root node of the *State-Action* tree and prune unnecessary tree nodes as in static environments.

We also delve into the time complexity of the S_A, SA_Q, and SA_L schedulers during the simulation phase. The S_A algorithm employs a random simulation policy with a time complexity of O(n), where n represents the depth of the simulation. The simulation policy of SA_Q and SA_L not only depends on the simulation depth but also on the efficiency of the function *getStateAction*() and their respective data structure updates. Specifically, the agent makes its action selection based on Q-table and *State-Action* tree, i.e. the time complexity of getting data from both the Q-table and the *State-Action* tree is O(1).[6] Additionally, the time complexity of the function *getStateAction*() is O(n), where n represents the number of steps executed by the agent during the selection phase. However, since this part of the computation is not included in the cyclic action selection of the simulation policy, the time complexity of SA_Q and SA_L is still O(n). In summary, despite the differences in the specific implementations of these schedulers, from the perspective of time complexity, it can be considered that in the simulation phase, the S_A, SA_Q and SA_L schedulers all share the same time complexity, i.e., O(n).

VI. EVALUATION

In this section, we evaluated the performance of SA_Q and SA_L in scenarios of increasing difficulty. In particular, we compare the performance of SA_Q and SA_L to the state-of-the-art scheduler S_A using sets of synthetic goal-plan trees [11].

A. EXPERIMENT SETUP

The synthetic goal-plan trees are randomly generated by a generator called genGPT [23]. This generator can generate random goal-plan trees with different shapes and properties by specifying a list of parameters including the depth of the tree, the plan branching factor (i.e., the maximum number of plans that can be used to achieve a goal), the goal branching factor (the maximum number of sub-goals a plan may have), the maximum number of actions in a plan and the number of environment variables that may appear in the tree. The first four parameters collectively influence the size and shape of the generated goal-plan trees. The number of environmental variables, on the other hand, will affect the possibility of having potential conflicts while achieving different goals in parallel, as all pre-conditions and post-conditions of actions are generated based on the environment variables. Similar to [11], each environment variable is modeled as a Poisson process with a specified mean value, thus allowing control over the frequency with which the environment changes the value of the variable. By controlling the parameters of the synthetic goal-plan tree generator and the dynamics of the environment, we can evaluate the performance of each scheduling algorithm under different conditions. The parameters for generating synthetic goal-plan trees in this paper are configured as follows:

- depth of the tree: 8
- number of plans to achieve a goal: 2
- number of subgoals in each plan: 1
- number of actions in each plan: 3
- number of environment variables: 60

In addition to the aforementioned parameter settings for the synthetic GPT generator, during the SA_Q update process, we do not have any preference between immediate and future rewards during the simulation. The discount factor γ is set to 0.5, which implies that the agent equally values immediate and future rewards. Similarly, the learning rate ω is also set to 0.5, indicating that new information and previously accumulated knowledge would be weighed equally when updating the Q-value.

We measure the performance of different schedulers based on two criteria: the number of goals achieved (the more, the better) and the computational overhead (the less, the better). Due to the fact that MCTS is an anytime algorithm, in most cases, the more time we give to the scheduler, the better performance the scheduler will have. We therefore come up with two different experiment settings. In the first set of experiments, the computational budget for each scheduler is specified by two variables α and β, where α represents the number of iterations to be performed, and β is the number of simulations performed in each iteration. The schedulers are given the same α and β values, and their performance is first evaluated based on the number of goals achieved. If the schedulers can achieve the same number of goals, we then evaluate their performance based on their computational overhead. This setting is essentially the same as those in [11].

Given the same α and β values, the computational overhead in our first set of experiments only reflects the time required for each simulation. In this paper, we focus more on how the proposed simulation policies will affect the overall efficiency of the scheduler rather than the time required to run a single simulation. That is, with the proposed simulation policy, each simulation may require additional computation time, however, the number of simulations required for the scheduler to produce the same results may also be significantly reduced. Therefore, in our second set of experiments, we give different schedulers the same computational time, and then compare the number of goals they can achieve.

1) STATIC ENVIRONMENT

In static environments, the mean of all Poisson processes was set to 0. In all experiments reported below, we report the average performance of each approach in 50 runs. We set the thresholds $\varepsilon = 0.1$ for both SA_Q and SA_L (a constant value in our proposed simulation policy controls the balance between selecting good actions based on historical data and exploring other potentially good actions). We then vary the number of goals (i.e. goal-plan trees) to be achieved from 1 to 15, increasing by 1 each time.

[6]Theoretically, the time complexity of *State-Action* tree fetching data is O(n), where n refers to the number of nodes in the *State-Action* tree. In order to improve the efficiency of the algorithm, we store the tree structure in the form of a table, so the time complexity of *State-Action* tree fetching data is the same as that of Q-table, both are O(1).

Experiment 1: In our first set of experiments, the computational budget for each scheduler is configured based on the value of α and β, i.e., the number of iterations to be performed and the number of simulations performed in each iteration.

First of all, the values of α and β are set to 100 and 10 respectively for all schedulers (which is the same as in [11]). As shown in Fig. 2, in terms of the number of goals achieved, the performance of all three schedulers improves as the total number of goals increases. The performance differences between the schedulers are negligible when the agent is given less than 3 goals. When the number of goals increases, we can see that both SA_L and SA_Q have a clear advantage over S_A, and the performance of SA_L is overall better than that of SA_Q. In cases where the agent is given more than 10 goals to achieve at the same time, SA_Q can achieve about 0.7 more goals compared to S_A on average, while SA_L can achieve 0.6 more goals compared to SA_Q. The superiority of SA_Q and SA_L over S_A is expected because both have optimized their simulation policy based on S_A, making full use of historical data to guide the simulations. SA_L outperforms SA_Q because of its superior calculation method, i.e., the method of calculating the average reward value is superior to the method of calculating the Q-value. More specifically, the calculation of the average reward value provides more reliable and accurate information compared to the Q-value.

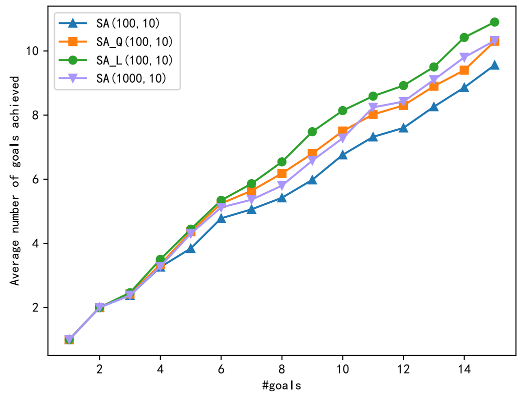

FIGURE 2. **The number of goals achieved with fixed α and β in static environment.**

In terms of the computational overhead, the performance (i.e., average computation time to achieve the given goals when α and β are fixed) of the three schedulers is shown in Fig. 3. As the number of goals to be achieved by the agent increases, the computational overhead for all three schedules increases. Given the same α and β values, S_A requires significantly less computational overhead compared to both SA_Q and SA_L. This result is not surprising, as SA_Q and SA_L require additional time to update the Q-table and *State-Action* tree based on state-action pairs. Overall, the performance of SA_Q is slightly better than that of SA_L because it requires less time to update the simulation results.

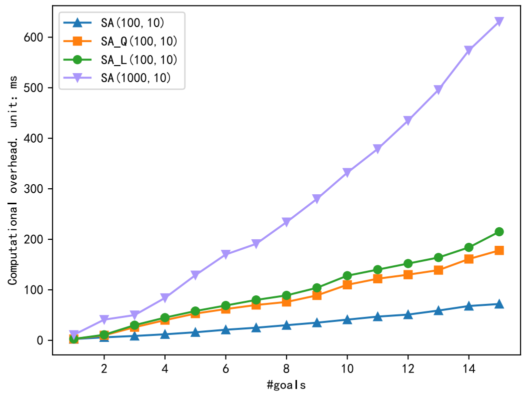

FIGURE 3. **Computational overhead with fixed α and β in static environment.**

Given the current results, it is difficult to decide if SA_Q and SA_L are more efficient than S_A, i.e., they can achieve more goals but require more time. Therefore, we conducted another set of experiments where we used S_A scheduler but with an increased α value, i.e., $\alpha = 1000$. The results are represented as purple lines in Fig. 2 and Fig. 3. As we can see in Fig. 2, in terms of the number of goals achieved, S_A with $\alpha = 1000$ performs nearly as well as SA_Q, although it is still slightly worse than SA_L. However, if we look at the computational overhead required for each scheduler in Fig. 3, we will see that S_A requires significantly more time to generate a similar performance compared to both SA_Q and SA_L.

Experiment 2: In the second set of experiments that follow, the computational budget of each scheduler is configured by a given computational time. More specifically, these three schedulers are all configured to perform 10 simulations ($\beta = 10$) per iteration. The number of iterations to be performed (α) is not pre-determined, instead, it is decided at run time according to the given computational time, i.e., the schedulers will stop immediately when they reach the given computational time. In the following experiment, we give different schedulers the same computation time and compare the number of goals they can achieve. Fig. 4 shows the performance of all three schedulers when the computation time for each top-level goal is set to 10 milliseconds. Given the same computational overhead, the number of goals each scheduler can achieve increases as the total number of goals given increases. When the number of goals allocated to the agent is less than 3, the differences between all three schedulers are negligible. The difference between SA_Q and SA_L is also almost negligible when the given number of goals is less than 5. However, at this point SA_Q and SA_L achieve about 0.5 more goals compared to S_A. As the number of top-level goals increases, we can see that both SA_Q and SA_L have a clear advantage over S_A, and SA_L is performing better compared to SA_Q. Overall, SA_Q can achieve about 0.6 more goals compared to S_A, while SA_L is able to achieve about 0.6 more goals compared to SA_Q on average. SA_L achieves more goals than SA_Q, as experiment 1 shows that a single simulation of SA_L incurs only slightly higher computational overhead than SA_Q. Therefore, under fixed computational overhead, the difference in the number of iterations performed between the two is not significant, e.g., in the case where the number of goals pursued by the agent is 10, SA_Q performs only about 12 more iterations than SA_L. However, since the method of calculating the average reward value by SA_L is superior to the method of calculating Q-value by SA_Q, SA_L results in a better performance of the agent while compensating for the effect of the gap in the number of iterations.

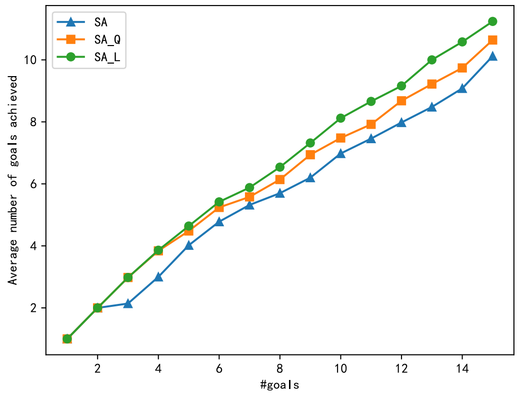

FIGURE 4. **Fixed computational overhead in static environment.**

2) DYNAMIC ENVIRONMENT

We then evaluate the performance of the proposed schedulers by setting a non-zero mean to the Poisson processes, which is essentially the same as those in [11]. That is, the value of the environment variables will be reversed after executing the selected action, based on the Poisson process. A larger mean value for the Poisson will cause more frequent changes in the environment. For each experiment reported below, we generate 10 goal-plan trees using the same parameters as in the static environment. We compare the performance of the schedulers by varying the mean of the Poisson process starting from 0.005 and increasing the value by 0.005 each time. Similar to the experiments in the static environment, we set the threshold $\varepsilon = 0.1$ for both SA_Q and SA_L. The results are the average performance of each approach in 50 runs.

Experiment 3: In our third set of experiments, we use α and β values to configure the computational budget as in experiment 1. The performances of the schedulers are again first evaluated based on the number of goals achieved, and then the computational time required. Similar to experiment 1, we set up four different settings, i.e., SA_A, SA_Q and SA_L schedulers with $\alpha = 100$ and $\beta = 10$, and the S_A scheduler with $\alpha = 1000$ and $\beta = 10$. The results are shown in Fig. 5. As we would expect, in terms of the number of goals achieved, the performances of all schedulers decrease as the value of the Poisson mean increases. Moreover, the differences between the schedulers also decrease as the Poisson mean becomes larger. For example, when the Poisson mean is set to 0.005, SA_Q is able to achieve about 0.7 more goals compared to S_A, while SA_L can achieve around 1.2 more goals compared to S_A. However, when the Poisson mean increases to 0.04, SA_Q and SA_L can only achieve 0.2 and 0.4 more goals compared to S_A respectively. The reason behind the scenes is that the unpredictable dynamic changes in the environment will hugely affect the overall performance of the schedulers, especially since some learned policies from previous simulations may be discarded due to unexpected changes in the environment. As the mean value gradually increases, environmental changes become more apparent. Consequently, frequent resets of the Q-table and *State-Action* tree are necessary, which is similar to having almost no prior experiences to draw from. As a result, the differences in the number of goals achieved by these three schedulers become smaller and eventually approach that of S_A. Overall, consistent with the results of experiment 1, SA_Q and SA_L consistently outperform S_A in terms of the number of goals achieved in almost all cases. In terms of the computational overhead, the results shown in Fig. 6 are similar to those in Fig. 3. That is, significantly more computational time will be required for S_A to achieve a similar level of performance as SA_Q and SA_L.

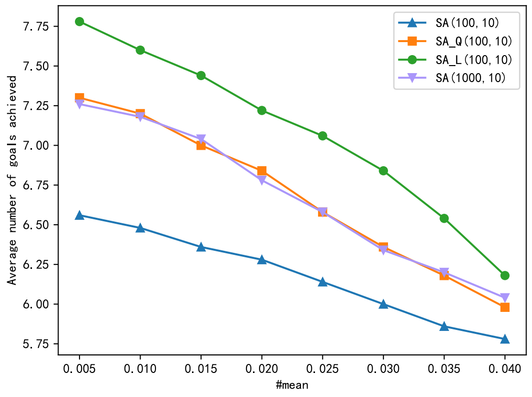

FIGURE 5. **The number of goals achieved with fixed α and β in dynamic environment.**

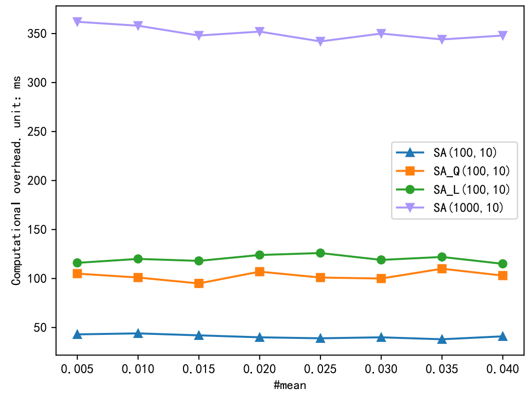

FIGURE 6. **Computational overhead with fixed α and β in dynamic environment.**

Experiment 4: Similar to experiment 2, in our last set of experiments, we evaluate and compare the performance of SA_L, SA_Q and S_A in the dynamic environment by giving them the same computational time. The results are shown in Fig. 7.

As can be seen, the number of goals achieved by three schedulers gradually decreases, as the Poisson mean increases. More specifically, as the mean value increases from 0.005 to 0.04, the differences between SA_L and S_A went down from 1 to 0.2. Moreover, the performance of SA_Q and S_A is almost identical when the Poisson mean increases to 0.04 if they are given the same computational time, i.e., the difference in the number of goals achieved between SA_Q and S_A is not more than 0.1. Overall, the results of the experiments are consistent with those in experiment 2, i.e., both SA_Q and SA_L can outperform S_A in most of the situations. Even in the worst case (i.e., the Poisson mean is set to 0.04), SA_Q and SA_L will not perform worse than S_A in a dynamic environment.

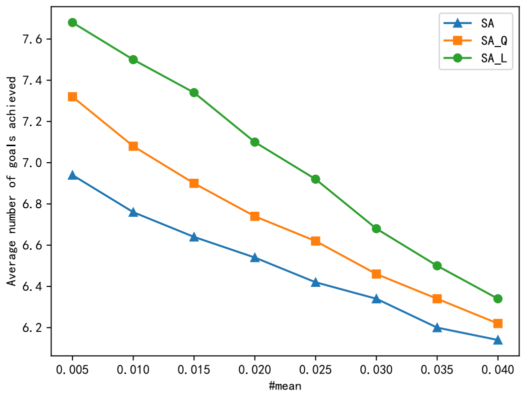

FIGURE 7. Fixed computational overhead in dynamic environment.

VII. RELATED WORK

In [24], Yao et al. have introduced the notion of quantitative summary information to estimate the likelihood of conflicts between an agent's intentions to avoid unnecessary simulations. In particular, the quantitative summary information was computed offline before the scheduling started, and it was then used to predict the probabilities of conflicts between different intentions at run time. In cases where the scheduling problem is extremely difficult or extremely easy, the simulation values can be directly predicted without running any real simulations. They have shown that the proposed approach can be used to improve the efficiency of the S_A scheduler, however, their solution works only for extreme cases. For cases where it is not easy to identify the solutions, the scheduler still needs to run random simulations as before.

In addition to the work discussed above, a number of other approaches to scheduling intentions to avoid conflicts have been proposed in the literature. As in [5], [6], [7], [8], and [25], intention selection is limited to the plan level, and the plans and sub-goals in a goal-plan tree are regarded as basic steps. In [25], Yao et al. first applied the MCTS algorithm to solve agent intention scheduling. They used a variant of MCTS called Single-Player MCTS [26] to schedule the intention of a single agent at the plan level. This work was then extended to the action level [11]. Shaw and Bordini have proposed approaches to intention selection based on Petri nets [27] and constraint logic programming [28]. Cope et al. [29] proposed a deep reinforcement learning-based planning method that builds on tree search by using neural networks to estimate the value function of each state to guide the search. There has also been work on avoiding conflicts in a multi-agent setting. Based on the previous work on single agent, Dann et al. [30] extended the MCTS algorithm to apply to multi-agent settings. Clement and Durfee [31], [32], [33] propose an approach to coordinating concurrent hierarchical planning agents using summary information and HTN planning.

In addition to the study of BDI agent in the intention progression problem, a number of approaches for combining BDI agent with learning for decision-making have been proposed in the literature. Singh et al. [34], [35] have proposed an approach that allows the agent to learn the probability of success for plans based on previous execution experiences. By using a probabilistic plan selection function, the agents can balance exploration and exploitation of their plans. Wan et al. [36] proposed a method to extend the BDI model with *Q*-learning which is one algorithm of reinforcement learning, and solve the problem that BDI agent can't decisions in dynamic and uncertain environments. Luna-Ramírez and Fasli [37] considered and developed intentional learning for agents within the Jason BDI framework, focusing on a plan acquisition strategy to address learning plans involving one action, sequences, or repeated actions that allow an agent to improve their behavior at run-time. Reference [38] deal with the issue of learning in multi-agent system (MAS) and proposed an extended BDI agent architecture with learning capabilities for MAS. The inductive approach of first-order *Logical Decision trees* is used by agent to learn when plans are successfully executed.

The above literature mainly discusses the application of BDI agents in the field of intention progression. BDI agents are regarded as representatives of rational agents, characterized by having clear goals and the ability to make decisions based on their beliefs, desires, and intentions to maximize the likelihood of achieving these goals. In contrast, non-rational agents may not adhere to optimization principles or logical reasoning in their decision-making process, but are likely constrained by various non-logical factors such as cognitive limitations, emotional influences, and social impacts. Yildiz et al. [39] studied binary opinion dynamics in a social network with stubborn agents and explored the role of stubborn agents in the process of opinion formation. Shang [40] introduced a general model for opinion formation in the averager-copier-voter network with non-rational agents, extending traditional consensus theory to accommodate a broader range of agent types. Furthermore, they proposed the averager-copier-voter filtering strategy that is capable of steering the robust opinion network toward a stochastic hybrid consensus in the sense of convergence in expectation.

VIII. CONCLUSION

In this paper, we present two novel approaches to improving the S_A 's simulation policy. The first approach is based on traditional Q-learning, while the second one uses a hierarchical structure called *State-Action* tree. Both approaches take into account the simulation results from previous simulations and learn how to favour more "promising" moves during the simulation phase. We then proposed SA_Q and SA_L, two variations of the S_A scheduler [11] based on the proposed approaches. We evaluated the performance of SA_Q and SA_L and compared it to the state-of-the-art S_A scheduler in both static and dynamic environments. The preliminary results indicate that our approaches have a clear advantage over the S_A scheduler in terms of efficiency in both static and dynamic environments.

As discussed in section III, we explore in depth the core strategies of the MCTS algorithm, including *tree policy*, *expansion* function, and *simulation policy*. The optimization of these strategies is crucial for enhancing the overall performance of the algorithm. In this paper, we primarily optimized the simulation policy, and may consider optimizing the *expansion* function in the future. The *expansion* function determines which nodes to add to the tree. By introducing constraints during the expansion phase, the algorithm can efficiently identify the most promising nodes, ultimately enhancing its overall performance. Considering the wide research prospects and practical application value of collaboration and competition issues in multi-agent system, another line of future work is to extend the approach proposed in this paper to multi-agent system.

REFERENCES

[1] A. S. Rao and M. P. Georgeff, "An abstract architecture for rational agents," in *Proc. 3rd Int. Conf. Princ. Knowl. Represent. Reasoning (KR)*, Cambridge, MA, USA. San Mateo, CA, USA: Morgan Kaufmann, Oct. 1992, pp. 439–449.

[2] M. Bratman, "Two faces of intention," *Phil. Rev.*, vol. 93, no. 3, p. 375, Jul. 1984.

[3] L. D. Silva, F. Meneguzzi, and B. Logan, "BDI agent architectures: A survey," in *Proc. 29th Int. Joint Conf. Artif. Intell.*, Jul. 2020, pp. 4914–4921.

[4] B. Logan, J. Thangarajah, and N. Yorke-Smith, "Progressing intention progression: A call for a goal-plan tree contest," in *Proc. 16th Conf. Auto. Agents MultiAgent Syst. (AAMAS)*, São Paulo, Brazil. New York, NY, USA: ACM, May 2017, pp. 768–772.

[5] J. Thangarajah and L. Padgham, "Computationally effective reasoning about goal interactions," *J. Automated Reasoning*, vol. 47, no. 1, pp. 17–56, Jun. 2011.

[6] J. Thangarajah, L. Padgham, and M. Winikoff, "Detecting & avoiding interference between goals in intelligent agents," in *Proc. 80th Int. Conf. Artif. Intell.*, Acapulco, Mexico. San Mateo, CA, USA: Morgan Kaufmann, Aug. 2003, pp. 721–726.

[7] J. Thangarajah, L. Padgham, and M. Winikoff, "Detecting & exploiting positive goal interaction in intelligent agents," in *Proc. 2nd Int. Joint Conf. Auto. Agents Multiagent Syst.*, Victoria, SA, Australia, Jul. 2003, pp. 401–408.

[8] J. Thangarajah, M. Winikoff, L. Padgham, and K. Fischer, "Avoiding resource conflicts in intelligent agents," in *Proc. 15th Eur. Conf. Artif. Intell. (ECAI)*, Lyon, France. IOS Press, Jul. 2002, pp. 18–22.

[9] M. Waters, L. Padgham, and S. Sardiña, "Evaluating coverage based intention selection," in *Proc. Int. Conf. Auto. Agents Multi-Agent Syst. (AAMAS)*, Paris, France, May 2014, pp. 957–964.

[10] M. Waters, L. Padgham, and S. Sardina, "Improving domain-independent intention selection in BDI systems," *Auto. Agents Multi-Agent Syst.*, vol. 29, no. 4, pp. 683–717, Jul. 2015.

[11] Y. Yao and B. Logan, "Action-level intention selection for BDI agents," in *Proc. 2016 Int. Conf. Auto. Agents Multiagent Syst.*, Singapore. New York, NY, USA: ACM, May 2016, pp. 1227–1236.

[12] Y. Yao, B. Logan, and J. Thangarajah, "Robust execution of BDI agent programs by exploiting synergies between intentions," in *Proc. 30th AAAI Conf. Artif. Intell.*, Phoenix, AZ, USA. Palo Alto, CA, USA: AAAI Press, Feb. 2016, pp. 2558–2565.

[13] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck, "Monte–Carlo tree search: A new framework for game AI," in *Proc. 4th Artif. Intell. Interact. Digit. Entertainment Conf.*, Stanford, CA, USA. Palo Alto, CA, USA: AAAI Press, Oct. 2008, pp. 216–217.

[14] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of Monte Carlo tree search methods," *IEEE Trans. Comput. Intell. AI Games*, vol. 4, no. 1, pp. 1–43, Mar. 2012.

[15] Y. Yao, L. de Silva, and B. Logan, "Reasoning about the executability of goal-plan trees," in *Proc. 4th Int. Workshop Eng. Multi-Agent Syst. (EMAS)*, Singapore, May 2016, pp. 181–196.

[16] Y. Yao, B. Logan, and J. Thangarajah, "Intention selection with deadlines," in *Proc. ECAI*. Amsterdam, The Netherlands: IOS Press, 2016, pp. 1700–1701.

[17] G. Chaslot, J.-T. Saito, B. Bouzy, J. Uiterwijk, and H. J. Van Den Herik, "Monte–Carlo strategies for computer go," in *Proc. 18th BeNeLux Conf. Artif. Intell.*, Namur, Belgium, 2006, pp. 83–91.

[18] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Mach. Learn.*, vol. 47, no. 2, pp. 235–256, 2002.

[19] M. N. Katehakis and A. F. Veinott, "The multi-armed bandit problem: Decomposition and computation," *Math. Oper. Res.*, vol. 12, no. 2, pp. 262–268, May 1987.

[20] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 1998.

[21] E. R. Gomes and R. Kowalczyk, "Dynamic analysis of multiagent Q-learning with ε-greedy exploration," in *Proc. 26th Annu. Int. Conf. Mach. Learn.*, Quebec, QC, Canada. New York, NY, USA: ACM, Jun. 2009, pp. 369–376.

[22] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, Feb. 2015.

[23] Y. Yao and D. Wu, "GenGPT: A systematic way to generate synthetic goal-plan trees," in *Proc. 9th Int. Workshop Eng. Multi-Agent Syst. (EMAS)*. Berlin, Germany: Springer, May 2021, pp. 373–380.

[24] Y. Yao, N. Alechina, B. Logan, and J. Thangarajah, "Intention progression using quantitative summary information," in *Proc. 20th Int. Conf. Auto. Agents Multiagent Syst.*, 2021, pp. 1416–1424.

[25] Y. Yao, B. Logan, and J. Thangarajah, "SP-MCTS-based intention scheduling for BDI agents," in *Proc. 21st Eur. Conf. Artif. Intell. (ECAI)*, Prague, Czech Republic. Amsterdam, The Netherlands: IOS Press, Aug. 2014, pp. 1133–1134.

[26] M. P. D. Schadd, M. H. M. Winands, M. J. W. Tak, and J. W. H. M. Uiterwijk, "Single-player monte-carlo tree search for SameGame," *Knowl.-Based Syst.*, vol. 34, pp. 3–11, Oct. 2012.

[27] P. H. Shaw and R. H. Bordini, "Towards alternative approaches to reasoning about goals," in *Proc. Int. Workshop Declarative Agent Lang. Technol.* Berlin, Germany: Springer, 2007, pp. 104–121.

[28] P. Shaw and R. H. Bordini, "An alternative approach for reasoning about the goal-plan tree problem," in *Languages, Methodologies, and Development Tools for Multi-Agent Systems*. Berlin, Germany: Springer, 2011, pp. 115–135.

[29] D. Cope, J. Svegliato, and S. Russell, "Learning to plan with tree search via deep RL," in *Proc. PRL Workshop Ser. Bridging Gap Between AI Planning Reinforcement Learn.*, 2023, pp. 285–293.

[30] M. Dann, J. Thangarajah, Y. Yao, and B. S. Logan, "Intention-aware multiagent scheduling," in *Proc. Adapt. Agents Multi-Agent Syst.*, 2020.

[31] B. J. Clement and E. H. Durfee, "Theory for coordinating concurrent hierarchical planning agents using summary information," in *Proc. 16th Nat. Conf. Artif. Intell., 11th Conf. Innov. Appl. Artif. Intell.*, Orlando, FL, USA. Cambridge, MA, USA: MIT Press, Jul. 1999, pp. 495–502.

[32] B. J. Clement and E. H. Durfee, "Performance of coordinating concurrent hierarchical planning agents using summary information," in *Proc. 4th Int. Conf. MultiAgent Syst.*, Jul. 2000, pp. 373–374.

[33] B. J. Clement, E. H. Durfee, and A. C. Barrett, "Abstract reasoning for planning and coordination," *J. Artif. Intell. Res.*, vol. 28, pp. 453–515, Apr. 2007.

[34] D. Singh, S. Sardiña, L. Padgham, and S. Airiau, "Learning context conditions for BDI plan selection," in *Proc. 9th Int. Conf. Auto. Agents Multiagent Syst. (AAMAS)*, Toronto, ON, Canada, May 2010, pp. 325–332.

[35] D. Singh, S. Sardina, and L. Padgham, "Extending BDI plan selection to incorporate learning from experience," *Robot. Auto. Syst.*, vol. 58, no. 9, pp. 1067–1075, Sep. 2010.

[36] Q. Wan, W. Liu, L. Xu, and J. Guo, "Extending the BDI model with Q-learning in uncertain environment," in *Proc. Int. Conf. Algorithms, Comput. Artif. Intell.*, Sanya, China. New York, NY, USA: ACM, Dec. 2018, pp. 33:1–33:6.

[37] W. A. Luna-Ramírez and M. Fasli, "Plan acquisition in a BDI agent framework through intentional learning," in *Proc. 15th German Conf. Multiagent Syst. Technol. (MATES)*, in Lecture Notes in Computer Science, vol. 10413, Leipzig, Germany. Berlin, Germany: Springer, 2017, pp. 167–186.

[38] A. Guerra-Hernández, A. E. F. Seghrouchni, and H. Soldano, "Learning in BDI multi-agent systems," in *Proc. 4th Int. Workshop Comput. Log. Multi-Agent Syst. (CLIMA)*, in Lecture Notes in Computer Science, vol. 3259, Fort Lauderdale, FL, USA. Berlin, Germany: Springer, Jan. 2004, pp. 218–233.

[39] E. Yildiz, A. Ozdaglar, D. Acemoglu, A. Saberi, and A. Scaglione, "Binary opinion dynamics with stubborn agents," *ACM Trans. Econ. Comput.*, vol. 1, no. 4, pp. 1–30, Dec. 2013.

[40] Y. Shang, "Hybrid consensus for averager–copier–voter networks with non-rational agents," *Chaos, Solitons Fractals*, vol. 110, pp. 244–251, May 2018.

CHENGCHENG SONG was born in Shandong, China, in 1999. She received the B.S. degree in software engineering, in 2021. She is currently pursuing the M.S. degree in software engineering with Zhejiang University of Technology, Zhejiang, China. Her main research interest includes multi-agent systems.

YUAN YAO (Member, IEEE) received the B.S. and Ph.D. degrees in computer science from the University of Nottingham, in 2012 and 2017, respectively. He is currently an Assistant Professor with the School of Computer Science, University of Nottingham Ningbo China. Prior to that, he was an Assistant Professor with the School of Computer Science and Technology, Zhejiang University of Technology, Hangzhou, China. His research interests include multi-agent systems, including agent reasoning, agent communications, and agent learning. Moreover, he is also interested in incorporating agent and the IoT techniques into real world applications.

SIXIAN CHAN received the Ph.D. degree from the College of Computer Science and Technology, Zhejiang University of Technology, in 2018. He is currently a Lecturer in computer science and technology with Zhejiang University of Technology. His research interests include image processing, machine learning, deep learning, and video tracking.