Proceedings of the Ninth Annual Conference on Advances in Cognitive Systems Submitted 9/2021; published 11/2021

Task Modifiers for HTN Planning and Acting

Weihang Yuan[1] WEY218@LEHIGH.EDU
Hector Munoz-Avila[1] HEM4@LEHIGH.EDU
Venkatsampath Raja Gogineni[2] GOGINENI.14@WRIGHT.EDU
Sravya Kondrakunta[2] KONDRAKUNTA.2@WRIGHT.EDU
Michael Cox[2] MICHAEL.COX@WRIGHT.EDU
Lifang He[1] LIH319@LEHIGH.EDU
[1]Department of Computing Science and Engineering, Lehigh University
[2]Department of Computer Science and Engineering, Wright State University

Abstract

The ability of an agent to change its objectives in response to unexpected events in the environment is a desirable trait. To provide such capability to hierarchical task network (HTN) planning, we propose an extension of the paradigm called *task modifiers* which are functions that receive a task list and a state and produce a new task list. We focus on a particular type of problems in which the planning and execution are interleaved and in which the ability to handle dynamic exogenous events is crucial. To demonstrate the efficacy of this approach, we evaluate the performance of agents equipped with simple heuristic task modifiers in two domains, including a marinetime simulation that differs substantially from traditional HTN domains.

1. Introduction

A key feature of cognitive systems is their reliance on structured representations of knowledge (Langley, 2012). A prominent structured representation is hierarchical task networks (HTNs) (Erol et al., 1994b). HTNs represent tasks in echelons that encode two explicit relations:

- Task-subtask relations: each task has a parent task (unless it is a root task) and one or more children (unless it is a leaf in the hierarchy).
- Order relations: sibling tasks at the root of the hierarchy or tasks that are children of the same parent have an ordering relation between them.

Because of these capabilities, HTN representations have been used in many applications (Nau et al., 2005) and adopted by cognitive architectures (Langley & Choi, 2006; Laird, 2019).

Another recurrent research topic in cognitive systems is goal reasoning, namely the idea of agents acting in an environment, self supervising the execution of their courses of action, and when discrepancies between the agent's own expectations and the actual observations from the environment are encountered, then new goals are generated as a result. For example, Cox et al. (2016) uses cause-effect relations $c \rightarrow d$, indicating the cause c (e.g., there is a mine field) for a discrepancy d

(e.g., a mine is encountered). When a discrepancy d is encountered the goal $\neg c$ is generated (e.g., remove the mine field) to obviate the discrepancy d.

Goal reasoning is associated with goals, conditions that can be ascertained if truth or false in the current state. For example, the condition "the agent encounters a mine" can be examined in the state. HTN planning, in contrast, reasons with tasks, activities to be performed that may or may not be goals[1]. However, a task in general have only an indirect meaning: it is defined by the methods that accomplished the task. They are (ambiguously) taught as "activities" to be performed. For instance, "seek for nearby mines", is an example of a task that doesn't represent an specific state condition and therefore it is not thought as a goal.

Despite these dichotomy between tasks and goals, there is no conceptual reason why goal reasoning capabilities couldn't be applied to systems reasoning with tasks. Specifically, the ability of an agent interleaving HTN planning and execution system to change its tasks as a result of changes in the environment. On these lines we present in this paper a modification of the SHOP HTN planning algorithm enabling the interleaving of planning and execution. More importantly, we add goal reasoning capabilities where the agent can modify its tasks as a result of discrepancies encountered during execution. SHOP is a widely adopted HTN planner, with many applications (Nau et al., 2005) including uses by cognitive architectures (e.g., Cox et al. (2016)).

2. Preliminaries

Our proposed approach is based on HTN planning, although we use more general definitions than the nomenclature in Nau et al. (1999) to better match our implementation.

A state variable describes an attribute of a domain world. For example, $loc(agent) = (0, 0)$ indicates that the agent is at location $(0, 0)$. A state s is a set of all state variables. The set of all states is denoted S.

Tasks symbolically represent activities in a domain. A task t consists of a name and a list of arguments and can be either primitive or compound. The set of all tasks is denoted T. A task list is a sequence of tasks $\tilde{t} = (t_1, \ldots, t_n)$. The set of all possible task lists is denoted $\tilde{T}$ (excluding the empty list).

A primitive task can be achieved by an action. An *action* is a 2-argument function $a : S \times \{t\} \rightarrow S \cup \{\texttt{nil}\}$,

$$a(s, t) = s', \tag{1}$$

where s is a state and t is a primitive task. If a is applicable to t in s, $s' \in S$. Otherwise, $s' = \texttt{nil}$.

A method describes how to decompose a compound task into subtasks. A *method* is a 2-argument function $m : S \times \{t\} \rightarrow \tilde{T} \cup \{\texttt{nil}\}$,

$$m(s, t) = \tilde{t},$$

where s is a state and t is a compound task. If m is applicable to t in s, $\tilde{t} \in \tilde{T}$. Otherwise, $\tilde{t} = \texttt{nil}$.

1. Some applications reason with goals only. For this reason, hierarchical goal network (HGN) planning was introduced. HGN is a variant of HTN where all tasks in the hierarchy are goals. Other approaches have combined tasks and goals (Nau et al., 2021).

An HTN planning problem is a triple $(s, \tilde{t}, D)$, where s is a state, $\tilde{t} = (t_1, \ldots, t_n)$ is a task list, and D is the set of all actions and methods. A plan is a sequence of actions. Solutions (plans) for HTN planning problems are defined recursively. A plan $\pi = (a_1, \ldots, a_m)$ is a solution for the HTN planning problem $(s, \tilde{t}, D)$ if one of the following cases is true:

1. If $\tilde{t} = \emptyset$, then $\pi = \emptyset$ (the empty plan).
2. If $\tilde{t} \neq \emptyset$:
   (a) If t_1 is primitive, a_1 is applicable (i.e., $a_1(s, t_1) \neq \texttt{nil}$), and $(a_2, \ldots, a_m)$ is a solution for $(a(s, t_1), (t_2, \ldots, t_n), D)$.
   (b) If t_1 is compound, there is an applicable method $m(s, t_1) \neq \texttt{nil}$, and π is a solution for $(s, (m(s, t_1), t_2, \ldots, t_n), D)$.

In Case 2 (a), after applying a_1, the remainder plan $(a_2, \ldots, a_m)$ is found to be a solution for the remaining tasks $(t_2 \ldots, t_n)$. In Case 2 (b), the compound task t_1 is replaced with $m(s, t_1)$, and π is found to be a solution for the new task list $(m(s, t_1), t_2, \ldots, t_n)$.

3. Task Modifiers

In this section, we describe an extension to HTN called task modifiers. We provide an example of a task modifier in a marine vehicle simulation domain. Then we describe an algorithm that integrates task modifiers and SHOP.

The motivation for task modifiers is to provide a mechanism that handles random events in some domains. Notably, this type of domain has the following characteristics:

- The agent observes an external environment and interacts with it through actions. The dynamics (state transitions) of the environment are defined by a set of Equations (1). In contrast, traditional HTN planning domains use operators to transform states.
- Actions are irreversible in the sense that the environment cannot revert to a previous state by editing state variables.
- The agent does not have full knowledge of the environment's dynamics. After an action is applied, the environment transitions to a new state. The agent needs to observe and acquire information about the state. This necessitates interleaved planning and execution of actions.
- The agent's observations are partial. The agent needs to make inferences about the values of variables not observed.
- The environment is episodic. A terminal signal is sent when an episode ends.

Traditional HTN planners recursively decompose high-level tasks into simpler ones. As discussed in Section 2, the agent's task list $\tilde{t} = (t_1, \ldots, t_n)$ can only be modified in one of two ways:

1. If t_1 is primitive and an applicable action for t_1 exists, the new task list is $(t_2, \ldots, t_n)$.

2. If t_1 is compound and an applicable method $m(s, t_1)$ exists, the new task list is $(m(s, t_1), t_2, \ldots, t_n)$.

Since the environment's dynamics are unknown to the agent, unexpected events may occur. For instance, the agent may encounter environmental hazards during a navigation task. This demands greater flexibility in terms of addition, deletion, and reordering of the tasks in $\tilde{t}$. For this reason, we introduce task modifiers that provide another way to modify a task list: replace $\tilde{t}$ with a new task list $\tilde{t}'$. A *task modifier* is a 2-argument function $TM : S \times \tilde{T} \to \tilde{T}$,

$$TM(s, \tilde{t}) = \tilde{t}'.$$

The definition of task modifiers is abstract. Any procedure that receives an observation s and a task list $\tilde{t}$ and outputs another task list $\tilde{t}'$ can be considered a task modifier. (When no changes are made to the task list, $\tilde{t} = \tilde{t}'$.) An abstract task modifier is used as part of an algorithm based on SHOP in Section 3.2. The implementation of two domain-specific task modifiers are described in Section 4.2.

3.1 Task Modifier Example

We show a use case of task modifiers in an underwater unmanned vehicle domain called Minefield, where the agent is tasked with maximizing the survival of some transport ships that traverse through an area. Further details of this domain are described in Section 4.1.

At the beginning of an episode (shown in Figure 1), 10 transport ships are placed on the left side of the central region. The agent, the pirate boat, and three fishing boats are randomly placed in the upper and lower regions. After 20 seconds, the transport ships start to move to the right side. The pirate continuously moves to random grid cells in the central region and places mines along the way. A transport ship is destroyed when it touches a mine. The agent has no direct knowledge of which boat is the pirate. The mines in a cell is automatically cleared as the agent moves to the cell. The initial task list is $(random\_moves)$. This task keeps the agent in constant patrol of the central region. The episode terminates when the remaining transport ships reach the right side or all the ships have been destroyed. The objective of the agent is to maximize the number of transport ships that survive.

The agent uses a task modifier that modifies the task list in response to dynamic events in the environment. For instance, after encountering a mine in a cell c, the agent decides to search nearby cells for more mines to clear:

$$(random\_moves) \Rightarrow (search\_near(c), random\_moves)$$

Alternatively, the agent may decide to pursue a suspect boat b and abandon other tasks:

$$(search\_near(c), random\_moves) \Rightarrow (follow(b))$$

To achieve a similar capability without a task modifier, each method has to be rewritten to handle dynamic events the same way a task modifier would. The code would be more complicated than just using a task modifier. Additionally, a method replaces a single task and is agnostic of other tasks in a task list. In contrast, a task modifier is more flexible and can change any part of the task list.

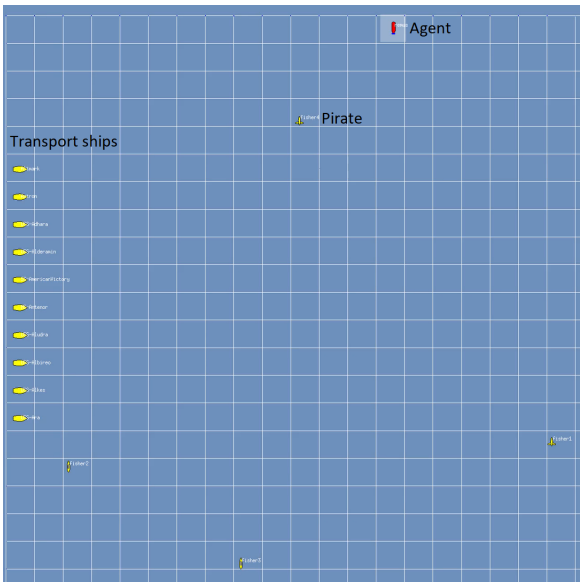


Figure 1: A view of the Minefield domain at the beginning of an episode.

3.2 Integrating Task Modifiers and SHOP

We now describe an algorithm that integrates SHOP with a task modifier and interleaves planning and execution. The pseudocode is shown in Algorithm 1. The task modifier used in the algorithm is abstract; the details of two domain-specific task modifiers are discussed in Section 4.2.

The differences between the original SHOP algorithm and our algorithm are underlined. At each time step, the agent observes the state of the environment and executes an action without full knowledge of state variables.

The algorithm starts with the procedure PLAN-ACT-TM (line 1) as an episode begins. The agent observes the current state s (line 2). The procedure SEEK-PLAN-ACT-TM is called (line 3).

SEEK-PLAN-ACT-TM (line 4) has a recursive structure similar to the HTN solution cases described in Section 2, but it does not maintains a plan because planning and execution is interleaved. If the current task list is empty or the episode terminates (line 5), the procedure returns (line 6). If the first task t in the task list is primitive (line 8) and an applicable action exists (line 9), the action is executed (line 10). Then the agent observes the next state s' (line 11). The task modifier receives s' and the remaining tasks R and updates the task list (line 12). The updated task list is passed to a recursive call of SEEK-PLAN-ACT-TM (line 13). If t is compound and an applicable method m is found (line 17), t is replaced by its reduction (subtasks) by m (line 18). When no applicable action or method is found, the procedure returns `nil` indicating a failure (lines 15 and 21).

The task modifier TM is only called (line 12) after an action is applied (line 10) but not after a method decomposes the first task in the task list (line 18). This is a design choice based on the experimental domains rather than a theoretical limitation. After an action is executed, unexpected changes (from the agent's perspective) might occur in the environment. In contrast, task decomposition is nearly instantaneous because the task list and the set of methods are internal to the agent.

Algorithm 1 SHOP with a Task Modifier

procedure PLAN-ACT-TM($\tilde{t}, D$)
  observe s
  return SEEK-PLAN-ACT-TM($s, \tilde{t}, D$)

procedure SEEK-PLAN-ACT-TM($s, \tilde{t}, D$)
  if $\tilde{t} = \emptyset$ **or** the episode terminates **then**
    return s
  $t \leftarrow$ the first task in $\tilde{t}$; $R \leftarrow$ the remaining tasks
  if t is primitive **then**
    if there is an action $a(s, t) \neq$ `nil` **then**
      apply a
      observe s'
      $R \leftarrow TM(s', R)$
      return SEEK-PLAN-ACT-TM(s', R, D)
    else
      return `nil`
  else
    for every method $m(s, t) \neq$ `nil` **do**
      $s \leftarrow$ SEEK-PLAN-ACT-TM($s, (m(s, t), R), D$)
      if $s \neq$ `nil` **then**
        return s
    return `nil`

4. Experiments

To demonstrate the usage of task modifiers, we tested our implementation in two domains.[2] Both domains have some of the characteristics described in Section 3 that are atypical of traditional HTN domains. The intent of the experiments is to provide a qualitative comparison between our implementation and two simple baselines.

4.1 Domains

Rainy Grid. In a 10×10 grid, the agent and a beacon randomly start at different locations and neither is at the exit, which is always in the bottom right corner. Each action produces a numerical reward. Rain occurs with a probability of p and affects an action's reward. The agent knows the locations of the beacon and the exit but not p. If the agent reaches the beacon, the rain stops until the end of the current episode. The episode terminates when the agent reaches the exit. The objective is to maximize the episodic cumulative reward. Rainy Grid has the following tasks:

- $move(dir)$. (Primitive) The agent moves one step right, up, left, or down. If it is not rainy, the action for this task has a reward of -1. If it is rainy, the action does nothing and has a reward of -5.
- $go\_to(dest)$. (Compound) $dest$ is either the beacon or the exit. The method for this task recursively decomposes it into $(move(dir), go\_to(dest))$, where dir is the direction toward $dest$.

Minefield. [3] Continuing the description in Section 3.1, the entire area is a 20×20 grid; the central region is 20×10. The pirate continuously selects a random cell in the central region and moves toward it. At each step, with a probability of p, the pirate places 20 mines according to a multivariate Gaussian distribution. Minefield has the following tasks (c is a grid cell and b is a boat):

- $move(c)$. (Primitive) The action for this task is applicable if the agent is adjacent to c. (The agent can move diagonally.)
- $arrest(b)$. (Primitive) The action for this task is applicable if the agent is adjacent to b. If b is the pirate, it ceases any activity for the rest of the episode. Otherwise, nothing happens.
- $random\_moves$. (Compound) This task is to randomly patrol the central region. The method for this task recursively decomposes it into $(move\_diag(c_1), random\_moves)$, where c_1 is a random cell.
- $move\_diag(c)$. (Compound) The task is to move along the shortest path to c. The method for this task recursively decomposes it into $(move(c_1), move\_diag(c))$, where c_1 is a cell adjacent to the agent and in the shortest path to c.

2. The code is available at `https://github.com/ospur/htn-tm`.
3. The domain was created using Mission Oriented Operating Suite Interval Programming (MOOS-IvP).

- $search\_near(c)$. (Compound) The task is to clear the mines in the adjacent cells of c. Note that the mines in a cell are removed once the agent reaches it. The method for this task decomposes it into $(move\_diag(c_1), \ldots, move\_diag(c_8))$, where $c_1, \ldots, c_8$ are the 8 cells adjacent to c in counterclockwise order.
- $follow(b)$. (Compound) The task is to follow b until the agent is in the same cell as b. The method for this task recursively decomposes it into $(move(c_1), follow(b))$, where c_1 is one step closer to the current location of b.

4.2 Implementation of Task Modifiers

Based on Algorithm 1, we created a modified version of Pyhop (a Python version of SHOP). Then we implement a task modifier for each domain. The following is a high-level description of the task modifiers and the intuition behind them.

Rainy Grid TM. The agent does not know the true rain probability. Instead, it assumes a rain probability p'. In our experiments, we set p' to 0.5. It computes the expected cost of a single move action $E(cost) = \frac{1+p'}{(1-p')}$. Then it computes the distance between its current location, the beacon, and the exit. Multiplying distance by the expected cost of a move action produce the expected cost of each task. The agent then decides whether to (1) directly go to the exit or (2) go to the beacon first and then the exit. The possible modifications are as follows:

1. $(go\_to(exit)) \Rightarrow (go\_to(beacon), go\_to(exit))$
2. $(go\_to(beacon), go\_to(exit)) \Rightarrow (go\_to(exit))$

Minefield TM. The Minefield TM modifies the task list as follows. Assume that the current task list is $(t_1, \ldots, t_n)$. There are several possible cases:

1. When the agent encounters one or more mines in a cell c, the task list is modified ($search\_near(c)$ is inserted):

$$(t_1, \ldots, t_n) \Rightarrow (search\_near(c), t_1, \ldots, t_n)$$

2. The agent update its estimate about which boat is the pirate and decide to follow it (suppose it is called $boat$). This modification has two steps: first, all previous $follow$ tasks are remove from the task list; then a new one is added. This only triggers if the pirate has not been arrested.

$$(t_1, \ldots, t_n) \Rightarrow (t'_1, \ldots, t'_m) \quad \text{(remove all } follow \text{ tasks so none appears on the RHS)}$$
$$(t'_1, \ldots, t'_m) \Rightarrow (follow(boat), t'_1, \ldots, t'_m) \quad \text{(add the new } follow\text{)}$$

3. When the agent is adjacent to a suspect boat b after following it and the pirate has not been arrested. The task $arrest(b)$ is added to the task list:

$$(t_1, \ldots, t_n) \Rightarrow (arrest(b), t_1, \ldots, t_n)$$

The identity of the pirate boat and the probability of it placing mines are both unknown, although the agent knows whether the pirate has been arrested. Under such circumstances, the behavior of the agent is to balance between two objectives:

- Clearing as many mines as possible: this objective has a direct impacts the number of transport ships that will survive. The few mines in the central area, the more transport ships is likely to survive.
- Arrest the pirate boat as soon as possible: when completed this objective prevents the pirate from placing more mines. However, solely focusing on this objective might result in many transport ships being destroyed in the process.

4.3 Baselines

Rainy Grid baselines. The purpose is to compare an agent that uses a task modifier with two baselines that have a fixed task list. The task list of Baseline 1 is $(go\_to(exit))$. The task list of Baseline 2 is $(go\_to(beacon), go\_to(exit))$. In other words, Baseline 1 always goes to the exit directly; Baseline 2 always goes to the beacon first (turning off the rain) and then the exit.

Minefield baselines. Since the domain contains many variables, it is useful to establish a minimal performance baseline when there is no agent at all. The purpose of this agentless baseline is to show that our task modifier agent indeed positively improves the survival of the transport ships. The other baseline is an agent that has random task modifier, which whenever invoked (Algorithm 1 line 12) inserts a random task to the beginning of the task list. This random baseline serves a similar purpose as the agentless baseline. It is used to show whether that our agent is performing better than just modifying the task list randomly.

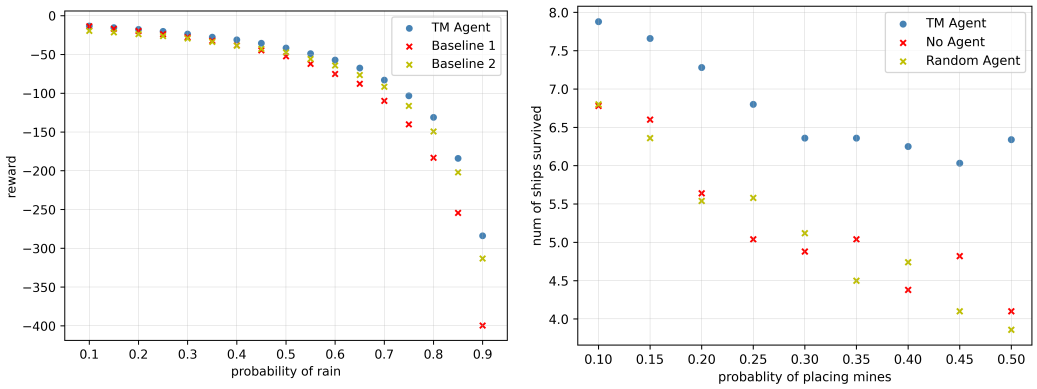
4.4 Results

Since each domain has an objective for the agent to achieve, the metric that indicates the performance of any given agent is based on that objective. In the Rainy Grid domain, each move by the agent has some cost associated with it, and therefore the metric is the total cost. For the minefield domain, the objective is to ensure as many transport ships survive as possible, and thus the number of surviving ships is the performance metric.

Figure 2 (a) shows the results of the task modifier agent and two baselines in the rainy grid domain. The vertical axis is the cumulative reward. The horizontal axis is the probability of rain. Each point is the average of 2000 runs. When the probability of rain is low, the task modifier agent performs similarly to the baselines. As the probability increases, the task modifier agent begins to outperform the baselines.

Figure 2 (b) shows that in the minefield domain, the task modifier agent outperforms the other two agents in all the probability configurations tested. The vertical axis is the number of transport ships that survive until the end of an episode. The horizontal axis is the pirate boat's probability of placing mines at each step. Each point is the average of 50 runs.

We conduct statistical significance tests on the data from both domains. Table 1 shows the results. The first number is the t-statistic and the second number is the p-value. For the rainy grid



(a) Rainy Grid

(b) Minefield

Figure 2: Comparison of the task modifier (TM) agent and baselines.

domain, due to the small difference in reward when the probability is low, we only run t-tests on the data where the probability of rain is above 0.6. We find statistically significant difference in the average reward between the task modifier agent and the baselines. For the minefield domain, we corroborate that the task modifier agent outperforms the other baselines through t-tests on the data where the probability of placing mines ranges from 0.2 (low) to 0.5 (high).

Rainy Grid	Probability of Rain			
	0.6	0.7	0.8	0.9
TM Agent and Baseline 1	14.10, 4.38e-44	14.31, 2.46e-45	16.74, 7.84e-61	17.55, 1.83e-66
TM Agent and Baseline 2	6.01, 2.01e-9	5.21, 2.04e-7	6.66, 3.05e-11	4.89, 1.03e-6
Minefield	Probability of Placing Mines			
	0.2	0.3	0.4	0.5
TM Agent and No Agent	3.86, 0.0002	3.03, 0.003	3.97, 0.0001	5.10, 1.63e-6
TM Agent and Random Agent	4.37, 3.10e-5	2.34, 0.021	3.25, 0.0016	5.78, 8.93e-8

Table 1: t-statistics and p-values on selected data of the task modifier agent and baselines.

5. Related Work

The idea of organizing actions hierarchically was proposed originally in Sacerdoti (1974). The Ordered Task decomposition implemented in SHOP and SHOP2 is the dominant HTN planning paradigm. It commits to a total order of the tasks unlike earlier versions (e.g., Wilkins (1999)), which combined partial-order planning into HTNs. SHOP2 enables partial-order between tasks in the methods, but when tasks are committed they need to be totally ordered. While loosing the flexibility of partial-order plan representations, Ordered Task Decomposition has resulted in significant running time speed gains, one of the key reasons why this paradigm has become dominant. In this work, we demonstrated goal reasoning capabilities for SHOP. The same design principles work on

SHOP2 as the only difference between SHOP and SHOP2 is how the first task is selected (Line 16 of the pseudocode); SHOP2 maintains a partially ordered list and hence it selects a task that has no preceding task (Nau et al., 2001).

Works on HTN planning have also studied its theoretical underpinnings including the result showing that HTN planning is strictly more expressive than STRIPS planning (Erol et al., 1994a). These representation capabilities has been exploited by cognitive architectures. For example, ICARUS learns hierarchies by using a teleoreactive process whereby gaps in planning knowledge triggers a procedure to learn targeted HTNs filling gaps in the hierarchy (Langley & Choi, 2006). SOAR integrates hierarchical execution and learning for dealing with dynamic environments (Laird & Rosenbloom, 1990). It dynamically selects actions based on its production memory. When a solution is successfully found, SOAR learns the decisions made that led to a successful solution. In our work, we focus on the hierarchical mechanism used in SHOP and enhance it to add goal reasoning capabilities.

Our work is also closely related to the actor view of planning (Ghallab et al., 2014), which emphasizes the need for interleaving of planning and execution. It advocates hierarchical online plan generation, where, quoting from Ghallab et al. (2014), *"the actor, refine, extend, update, change, and repair its plans throughout the acting process"*. This builds on a long tradition of studies interleaving planning and execution. Indeed, as earlier in Fikes & Nilsson (1971), proposes execution strategies for plans including adding inhibitors to ensure that invalid actions are not executed. Cognitive architectures including ICARUS, SOAR and MIDCA use a variety of mechanisms to identify gaps in their planning knowledge detected during plan execution and learn to fill those gaps. Other works combining HTN planning and execution include Sirin et al. (2004) which uses HTN planning to generate semantic web service composition plans. Sirin et al. (2004) uses methods to decompose task into subtasks; in our work we use methods in the same way. However, we task modifiers to replace task sequences in the HTN.

Our work is also related to task insertion (e.g., Xiao et al. (2020)). Task insertion is used in domains where the HTN methods are incomplete. Primitive tasks are inserted in the HTN to fill gaps in the methods. In our work task modifiers are used to react to changes in the environment but not necessarily to fill gaps in the HTN methods.

Our work is also related to replanning. Replanning is needed when a failure in the execution of the current plan is encountered in an state s_{fail} that prevents it from continuing the execution of the portion of the plan π_{yet} yet to be executed. For instance, Fox et al. (2006) examines two plan completion strategies: adapting π_{yet} or generating a new plan from the scratch starting from state s_{fail}. In general, adapting π_{yet} is known to be computationally harder than planning from scratch (Nebel & Koehler, 1995). Warfield et al. (2007) presents a plan adaptation algorithm for HTNs. The difference between replanning and our work is that goal reasoning changes the tasks including the input task list. Whereas in replanning the overarching goals (and in particular for HTNs the input task list) remains the same.

POMDPs is a planning paradigm for partially observable states. They enable to plan in advance for any contingency the agent may encounter (Kaelbling et al., 1998). In its basic formulation, the agent has knowledge indicating the probability, $O(s, a, o)$, of making observation o when executing action a in state s. So for example, the agent might know the layout of a labyrinth it is navigating but

not its own exact location within. However, it has partial information about its own whereabouts. For instance, when the agent moves forward it may encounter a red wall (i.e., an observation o) and as a result it will know it finds itself in anyone of 4 locations in the labyrinth with a probability $1/4$. Using this and other pieces of information, a POMDP agent can plan for every circumstance ahead of time. Goal reasoning in general and this work in particular are motivated for situations when the agent encounters unforeseen situations (e.g., no prior knowledge of $O(s, a, o)$). However, in our work the goals may change due to unforeseen situations and the agent needs to replan accordingly.

6. Conclusions

In this paper we introduced the notion of task modifier. A task modifier changes the task list as a reaction to unexpected observations in the environment. We show how to enhance the SHOP HTN planning algorithm so that it can exploit the task modifier thereby endowing it with goal reasoning capabilities. The resulting algorithm interleaves planning and execution. We implemented this algorithm and demonstrated it on two very distinct domains: in one the agent is controlling an UUV aiming at protecting transport ships and in the other one the agent is navigating in an stochastic environment.

For future work, we want to create a task modifier procedure that is more domain-independent. In addition to being a function, this version of a task modifier has access to a set of modifications. A modification consists of a task list and a set of conditions; it is applicable to a task list given an observation if the conditions holds. Applying the modification results in a new task list. The task modifier will select the appropriate modification based on some criteria. Learning of the criteria is potentially another problem to solve.

Acknowledgements

This research is supported by the Office of Naval Research grants N00014-18-1-2009 and N68335-18-C-4027 and the National Science Foundation grant 1909879 and Independent Research and Development (IR/D) Plan. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

Cox, M., Alavi, Z., Dannenhauer, D., Eyorokon, V., Munoz-Avila, H., & Perlis, D. (2016). Midca: A metacognitive, integrated dual-cycle architecture for self-regulated autonomy. *Proceedings of the AAAI Conference on Artificial Intelligence*.

Erol, K., Hendler, J., & Nau, D. S. (1994a). Htn planning: Complexity and expressivity. *AAAI* (pp. 1123–1128).

Erol, K., Hendler, J. A., & Nau, D. S. (1994b). Umcp: A sound and complete procedure for hierarchical task-network planning. *Aips* (pp. 249–254).

Fikes, R. E., & Nilsson, N. J. (1971). Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, *2*, 189–208.

Fox, M., Gerevini, A., Long, D., & Serina, I. (2006). Plan stability: Replanning versus plan repair. *ICAPS* (pp. 212–221).

Ghallab, M., Nau, D., & Traverso, P. (2014). The actor's view of automated planning and acting: A position paper. *Artificial Intelligence*, *208*, 1–17.

Kaelbling, L. P., Littman, M. L., & Cassandra, A. R. (1998). Planning and acting in partially observable stochastic domains. *Artificial intelligence*, *101*, 99–134.

Laird, J. E. (2019). *The soar cognitive architecture*. MIT press.

Laird, J. E., & Rosenbloom, P. S. (1990). Integrating, execution, planning, and learning in soar for external environments. *AAAI* (pp. 1022–1029).

Langley, P. (2012). The cognitive systems paradigm. *Advances in Cognitive Systems*, *1*, 3–13.

Langley, P., & Choi, D. (2006). A unified cognitive architecture for physical agents. *Proceedings of the National Conference on Artificial Intelligence* (p. 1469). Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999.

Nau, D., Au, T.-C., Ilghami, O., Kuter, U., Wu, D., Yaman, F., Munoz-Avila, H., & Murdock, J. W. (2005). Applications of shop and shop2. *IEEE Intelligent Systems*, *20*, 34–41.

Nau, D., Bansod, Y., Patra, S., Roberts, M., & Li, R. (2021). Gtpyhop: A hierarchical goal+ task planner implemented in python. *ICAPS Workshop on Hierarchical Planning (HPlan)*.

Nau, D., Cao, Y., Lotem, A., & Munoz-Avila, H. (1999). Shop: Simple hierarchical ordered planner. *Proceedings of the 16th international joint conference on Artificial intelligence-Volume 2* (pp. 968–973).

Nau, D., Munoz-Avila, H., Cao, Y., Lotem, A., & Mitchell, S. (2001). Total-order planning with partially ordered subtasks. *IJCAI* (pp. 425–430).

Nebel, B., & Koehler, J. (1995). Plan reuse versus plan generation: A theoretical and empirical analysis. *Artificial intelligence*, *76*, 427–454.

Sacerdoti, E. D. (1974). Planning in a hierarchy of abstraction spaces. *Artificial intelligence*, *5*, 115–135.

Sirin, E., Parsia, B., Wu, D., Hendler, J., & Nau, D. (2004). Htn planning for web service composition using shop2. *Journal of Web Semantics*, *1*, 377–396.

Warfield, I., Hogg, C., Lee-Urban, S., & Munoz-Avila, H. (2007). Adaptation of hierarchical task network plans. *FLAIRS conference* (pp. 429–434).

Wilkins, D. E. (1999). Using the sipe-2 planning system. *Artificial Intelligence Center, SRI International, Menlo Park, CA*.

Xiao, Z., Wan, H., Zhuo, H. H., Herzig, A., Perrussel, L., & Chen, P. (2020). Refining htn methods via task insertion with preferences. *Proceedings of the AAAI Conference on Artificial Intelligence* (pp. 10009–10016).