ICAPS Workshop on Hierarchical Planning (HPlan), July 2021

Integrating Planning and Acting With a Re-Entrant HTN Planner

Yash Bansod[1], Dana Nau[1,2], Sunandita Patra[2], Mak Roberts[3]
[1]Institute for Systems Research and [2]Dept. of Computer Science, Univ. of Maryland, College Park, MD, USA
[3]The U.S. Naval Research Laboratory, Code 5514, Washington, DC, USA
{yashb, nau, patras}@umd.edu, {mak.roberts}@nrl.navy.mil

Abstract

A major problem with integrating HTN planning and acting is that, unless the HTN methods are very carefully written, unexpected problems can occur when attempting to replan if execution errors or other unexpected conditions occur during acting. To overcome this problem, we present a re-entrant HTN planning algorithm that can be restarted for replanning purposes at the point where an execution error occurred, and an HTN acting algorithm that can restart the HTN planner at this point. We show through experiments that our algorithm is an improvement over a widely used approach to planning and control.

1 Introduction

HTN planners use *descriptive models* of actions tailored to compute the next states in a state transition system efficiently. In most cases[1] they assume a world that is closed, static, and deterministic. However, executing the plan in open, dynamic, and nondeterministic environments, characteristic of many practical problems, generally leads to failure. The planning domain will rarely be an entirely accurate model of the actor's environment, and execution of the plan may fail due to (i) failure in execution of actions, (ii) occurrence of unexpected events, (iii) because the planning was solved with incorrect or partial information.

Plans are needed for deliberative acting, but are not sufficient for it (Pollack and Horty 1999). Many deliberative acting approaches seek to combine the descriptive models used by the planner with the *operational models* used by the actor (Ingrand and Ghallab 2017). In contrast, others seek to directly integrate planning and acting using operational representations (Patra et al. 2019, 2020).

An early version of HTN planning was the Simple Hierarchical Ordered Planner (SHOP) (Nau et al. 1999), and its successors SHOP2 and SHOP3 (Nau et al. 2003; Goldman and Kuter 2019). SHOP and its successors are written in the LISP programming language, which limits its adoption.

Python is a much more widely adopted programming language used by roboticists, game developers, machine learning engineers, and AI engineers. The Pyhop planner (Nau 2013a,b) adapts the SHOP planning algorithm so that methods and actions are written directly in Python. GTPyhop (Nau et al. 2021), a recent extension to Pyhop, combines both HTN planning and hierarchical goal-network (HGN) planning (Shivashankar et al. 2012).

One difficulty with integrating acting with HTN planning is responding to action failures at execution time. If one tries to replan by calling the HTN planner with the new current state but the same task as before, unfortunate results can occur (Section 5.1). In this paper we describe a way to overcome that problem. Our primary contributions are:

1. We describe IPyHOP, a planner that can respond to plan-execution failures by resuming the planning process at the point where the failure occurred. IPyHOP's planning algorithm is based on GTPyhop (Nau et al. 2021), but with the following key changes: it uses iteration rather than recursion, and it preserves the hierarchy in the planning solution and returns a solution task network rather than a simple plan. Thus, if an unexpected problem occurs during plan execution, the actor can call IPyHOP with a pointer to the point in the task network where the execution failure occurred, and IPyHOP can resume planning from that point onward.
2. Inspired by the RAE actor in (Ghallab, Nau, and Traverso 2016, Chapter 3), we provide a new acting algorithm, Run-Lazy-Refineahead, that integrates efficiently with IPyHOP. Run-Lazy-Refineahead calls IPyHOP to get a solution task network and executes the actions in the task network by sending them to its execution platform. If an execution failure occurs, it gives IPyHOP a pointer to where the failure occurred and requests replanning.

After discussing related work in Section 2, Section 3 explains our notation and briefly provides background on Pyhop and GTPyhop. Section 4 explains the HTN planning in IPyHOP. Section 5 explains the Run-Lazy-Lookahead and Run-Lazy-Refineahead algorithms and how IPyHOP can be used in integrated HTN planning and acting or deliberative HTN acting. Sections 6 and 7 describe an experimental domain for HTN planning and experiments that compare Run-Lazy-Lookahead and Run-Lazy-Refineahead algorithms for deliberative HTN acting. Finally, Section 8 summarizes our work and discusses limitations and some avenues for future research.

[1] There are some exceptions, e.g., (Kuter and Nau 2005; Hogg, Kuter, and Muñoz-Avila 2009; Chen and Bercher 2021).

2 Related Work

AI planning. HTN planning is a widely adopted approach to AI planning in the gaming industry (Neufeld et al. 2017). One of the first HTN planners was Nets of Action Hierarchies (NOAH) (Sacerdoti 1975). Since then, numerous HTN planners have been developed. Some of the best-known ones are Nonlin (Tate 1977), System for Interactive Planning and Execution (SIPE) and SIPE-2 (Wilkins 1990), Open Planning Architecture (O-Plan) (Currie and Tate 1991) and its successor O-Plan2 (Tate, Drabble, and Kirby 1994), Universal Method Composition Planner (UMCP) (Erol 1996), SHOP (Nau et al. 1999) and its successor SHOP2, and SHOP3 (Nau et al. 2003; Goldman and Kuter 2019), and SIADEX (Castillo et al. 2005). Additionally, there are various HTN planners like Simple Hierarchical Planning Engine (SHPE) (Menif, Jacopin, and Cazenave 2014) that are specifically developed for AI planning in video games.

A wide body of literature also exists on Monte Carlo tree search based planning in games. Monte Carlo tree search refers to simulated execution (Feldman and Domshlak 2013, 2014), sampling outcomes of action models (Yoon, Fern, and Givan 2007; Teichteil-Koenigsbuch, Infantes, and Kuter 2008), and hindsight optimization (Yoon et al. 2008).

Planning and acting. Musliner et al (2008) propose a way to do online planning and acting. The old plan is executed repeatedly in a loop while the planner synthesizes a new plan (which the authors say can take a significant amount of time), and the new plan is not installed until planning has been finished. This way of repeated planning and acting is similar to the Run-Concurrent-Lookahead algorithm defined in (Ghallab, Nau, and Traverso 2016, Chapter 2).

Other similar algorithms, e.g., Run-Lookahead and Run-Lazy-Lookahead, are also defined in (Ghallab, Nau, and Traverso 2016, Chapter 2). Here Lookahead is any *online* planning algorithm. Each time Run-Lookahead calls the Lookahead planner, it performs only the first action of the plan that Lookahead returned. This way of execution is effective, for example, in unpredictable or dynamic environments in which some of the states are likely to be different from what the planner predicted. In contrast, Run-Lazy-Lookahead executes each plan as far as possible, calling Lookahead again only when the plan ends, or a plan simulator says that the plan will no longer work properly.

BDI Architectures. BDI (Belief-Desire-Intention) architectures (De Silva and Padgham 2005; Bauters et al. 2014; Yao et al. 2021; Sardina, De Silva, and Padgham 2006) have some similarity to our work, but BDI systems are mostly reactive. They differ from us with respect to their primitives as well as their methods or plan-rules. In general, BDI systems will not replan, but they will select and execute an untried method when failure occurs. Some BDI approaches, e.g., (Yao et al. 2021) can also replan, but their agent model is non-hierarchical. (Clement, Durfee, and Barrett 2007) integrates BDI architecture with hierarchical agent models for temporal planning and coordination in multi-agent systems.

3 Background: Pyhop and GTPyhop

In this paper an HTN *planning domain* is defined as a pair $\Sigma = (O, M)$, and an HTN *planning problem* is defined as a 4-tuple $\mathcal{P} = (s_0, w, O, M)$, where s_0 is the initial state, w is the inital task network, O is a set of operators, and M is a set of HTN methods. For details, see (Bansod 2021). We assume that primitive tasks can be directly executed by the execution engine but non-primitive tasks need refinement before execution. We also assume that the planning domain is deterministic and fully observable, but the execution environment is nondeterministic—hence a solution plan returned by the planner might not always work correctly at execution time.

GTPyhop (Nau et al. 2021) is a domain-independent Goal Task Network (GTN) planning system written in Python. GTPyhop is a progressive totally-ordered GTN planner i.e. it plans for a sequence of tasks and goals in the same order that they will later be executed. This behavior helps avoid some of the goal-interaction issues that arise in other HTN planners, making the planning algorithm relatively simple. The planning algorithm is sound and complete over a large class of problems. Since GTPyhop knows the complete world-state at each step of the planning process, it can use highly expressive domain representations.

GTPyhop uses recursion for task and goal refinement. Writing the algorithm as a recursive algorithm is intuitive, and it follows the description of HTN planning algorithm in many texts. The algorithm is also simple to implement, and the recursion stack efficiently handles the refinement and backtracking during task planning.

Like most HTN planners, GTPyhop has two limitations that limit its ability to do effective replanning. First, the use of recursion prevents the code from being re-entrant. If it is necessary to replan because of an action failure, the only alternative is to call GTPyhop again, which can lead to incorrect results (see Section 5.1). Second, GTPyhop returns a plan π, but not the refinement tree that produced π. In order to replan if an action failure occurs, it is necessary for a planner to know not only the action that failed but what tasks it was trying to achieve at that point in the planning process. That requires a copy of the refinement tree.

4 HTN Planning in IPyHOP

IPyHOP overcomes the limitations of GTPyhop in two ways. First, it uses an iterative tree traversal procedure for task refinement, with the refinement and backtracking done by tree traversal algorithms. This supports considerably more control over how the algorithm refines tasks. Second, it accepts a (partial) task tree and returns the entire solution task network. This change supports adding hierarchical knowledge for the replanning process. IPyHOP implements GTN planning like GTPyhop. However, in this paper, we discuss only the HTN planning functionality of IPyHOP.

Let u represent a grounded task node. Then, $\mathit{task}(u)$ defines the grounded task $t = t(r_1, ..., r_k)$ corresponding to u. $\mathit{refined}(u) \in \{\mathit{true, false}\}$ represents if the node has been refined. $\mathit{operator}(u)$ represents the operator $o \in O$ that is relevant to task t if the task was primitive. $\mathit{visited}(u) \in \{\mathit{true, false}\}$ represents if the node has been visited. $\mathit{state}(u)$

Algorithm 1: HTN Planning in IPyHOP.

```
IPyHOP($s, w, O, M$):
p ← $root(w)$
while true do
    $u$ ← first_unrefined_bfs_successor($w, p$)
    if $u = \emptyset$ then
        if $p$ = $root(w)$ then
            break
        else
            $p \leftarrow parent(p)$
            continue
    $t \leftarrow task(u)$
    if $t$ is primitive then
        $o \leftarrow operator(u)$        \\ here $o \in O$
        $s' \leftarrow o(s, r_1, ..., r_k)$
        if $s'$ is valid then
            $s \leftarrow s'$
            $refined(u)$ ← true
        else
            $w, u$ ← backtrack($w, u$)
            $p \leftarrow parent(u)$
    if $t$ is non-primitive then
        if $visited(u)$ then
            $s \leftarrow state(u)$
        else
            $visited(u) \leftarrow true$
            $state(u) \leftarrow s$
        foreach $m \in methods(u)$ where $m \in M$ do
            $t' \leftarrow m(s, r_1, ..., r_k)$
            $methods(u) \leftarrow methods(u) \backslash m$
            if $t'$is valid then
                $refined(u) \leftarrow true$
                add_nodes($u, t'$)
                $p \leftarrow u$
                break;
        if not $refined(u)$ then
            $w, u$ ← backtrack($w, u$)
            $p \leftarrow parent(u)$
return $w$
```

represents the state when the node was first visited. And $methods(u)$ represents the methods applicable to the task t that haven't been used for refinement of u, given that the task is non-primitive.

Algorithm 1 is IPyHOP's HTN planning algorithm. The first_unrefined_bfs_successor(w, p) returns the first unrefined node found during a breadth first search in w, starting from node p. In IPyHOP, backtrack(w, u) is a subroutine (see Algorithm 2) that modifies the task network given that refinement of node u failed. W_p is a list of nodes in a depth first search pre-ordering in w, starting from node p. After backtracking, the non-primitive task node u', the node refined before the current task node u, is again marked for refinement. The add_nodes(u, t') subroutine adds the sub-tasks t' as nodes to the refined node u.

Algorithm 2: IPyHOP Backtracking.

```
backtrack($w, u$):
$p \leftarrow parent(u)$
$W_p$ ← dfs_preorder_nodes($w, p$)
foreach $v \in reversed(W_p)$ do
    $refined(v) \leftarrow false$
    if $v$ is non-primitive then
        $W_v \leftarrow descendants(v)$
        $w \leftarrow w \backslash W_v$
        return $w, v$
$w \leftarrow \{root(w)\}$
return $w, root(w)$
```

5 Integrating IPyHOP with an Actor

As explained in Section 2, a popular way of integrating a planner and an actor is by using algorithms like Run-Lazy-Lookahead. In Section 5.1 we describe the Run-Lazy-Lookahead algorithm and some of its features. We explain its use in deliberative HTN acting and point to some of its limitations. In Section 5.2 we describe the Run-Lazy-Refineahead algorithm for deliberative HTN acting.

5.1 Where Run-Lazy-Lookahead Fails

The Run-Lazy-Lookahead algorithm, previously introduced in Ghallab, Nau, & Traverso (Chapter 2 2016) is a deliberative acting algorithm. It executes each plan π as far as possible, calling Lookahead again only when π ends or a plan simulator says that π will no longer work properly. This way of execution can help in environments where it is computationally expensive to call Lookahead, and the actions in π are likely to produce the predicted outcomes. It can also use a plan simulator, which may use the planner's prediction function γ or may do a more detailed computation (e.g., a physics-based simulation, a Monte-Carlo simulation, et cetera.) that would be too time-consuming for the planner to use. The simulator should return failure if its simulation indicates that π will not work correctly. For example, if it finds that an action in π will have an unsatisfied precondition, or if the simulation indicates that the π will not achieve the goal g when it is supposed to.

We can use IPyHOP as the Lookahead planner in Run-Lazy-Lookahead to integrate HTN planning and acting. However, this repeated planning and acting procedure does not work well with HTN planners. The problem can be visualized with the following abstract example.

Example 1. Suppose we want to plan for a task network consisting of two tasks $t1$ and $t2$. Let there be two methods $m1\_t1$ and $m2\_t1$ that are applicable to $t1$. And two methods $m1\_t2$ and $m2\_t2$ that are applicable to $t2$. Let primitive tasks be represented in syntax $o\langle i\rangle$, ex. $o1, o2$ et cetera. Let $m1\_t1$ refine $t1$ into $o1$ and $o2$. Let $m2\_t1$ refine $t1$ into $o3$, $o4$, and $o5$. Let $m1\_t2$ refine $t2$ into $o4$, $o5$ and $o6$. And let $m2\_t2$ refine $t2$ into $o7$ and $o8$. Also, for the sake of this example assume that all tasks, methods, and the operators defined here are grounded. These individual refinements can be visualized in Figure 1. We will assume that all the methods and operators have no pre-conditions and all are applicable

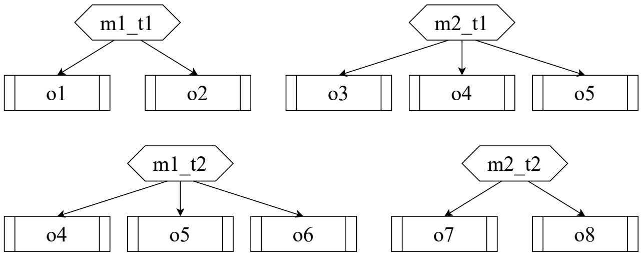


Figure 1: Refinement of task $t1$ using $m1\_t1$ and $m2\_t1$. And refinement of task $t2$ using $m1\_t2$ and $m2\_t2$. (Example 1).

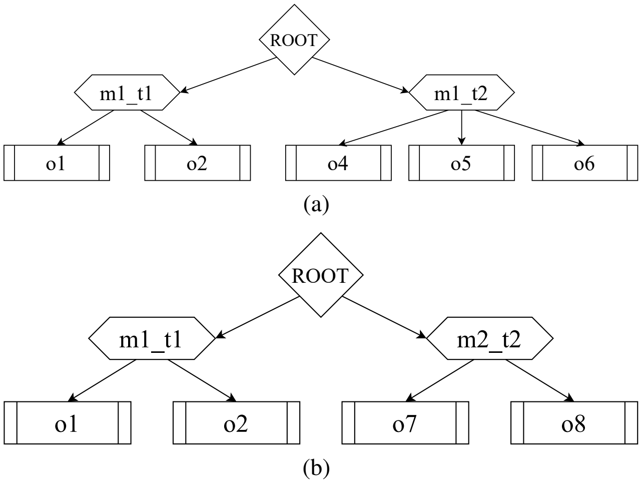


Figure 2: Task network visualizations: (a) After first planning attempt. (b) Re-planning after failure in execution of o6.

anytime in the planning process. Also, assume that the HTN planner always prioritizes refinement of tasks using the first method over second.

The solution tree that IPyHOP will return is visualized by Figure 2(a). This solution implies that the plan represented in the form of a primitive task sequence will be $\pi = \langle o1, o2, o4, o5, o6\rangle$. The primitive task sequence is found by performing a Depth First Search (DFS) tree traversal on the solution tree. Let us assume that while executing this plan, $o6$ nondeterministically fails. We update our model of $o6 \in O$ (if required) used by the planner and perform re-planning again. The new solution tree that IPyHOP will return is visualized by Figure 2(b). The actor will now execute the plan $\pi = \langle o1, o2, o7, o8\rangle$. This means that the action sequence executed by our actor is $\alpha = \langle o1, o2, o4, o5, o6, o1, o2, o7, o8\rangle$, when in fact it should have been $\alpha = \langle o1, o2, o4, o5, o6, o7, o8\rangle$ for the given scenario. This action sequence was executed because we did re-planning for the completed task $t1$ along with the failed task $t2$. ■

Technically, it is possible to prevent degenerate executions like in Example 1 from happening by cleverly designing methods that consider failures or having some flags in the state that gets modified. However, as the complexity of the task network increases, this approach quickly becomes infeasable. One of the most significant limitations of HTN planning is the substantial domain engineering effort required in writing HTN methods. Domain authoring is especially hard because the HTN formalism requires users to provide methods to cover every possible scenario that the agent could encounter. If the HTN planner finds itself in a situation the user had not anticipated, it will behave unexpectedly or fail without returning a solution. Moreover, there are many scenarios where it is impossible to account for such occurrences while authoring the domain.

5.2 Run-Lazy-Refineahead

The problems with Run-Lazy-Lookahead occur due to incompatibilities between the definition of the Lookahead planner and the definition of an HTN planner. The signature of a Lookahead planner is (Σ, s, g), whereas the signature of HTN planners is (s, w, O, M). However, the goal g and task network w are notably different. The goal for a planner might stay unchanged as the plan is executed. However, the task network is constantly modified. Replacing the Lookahead planner with IPyHOP leads to repeated planning for some of the completed tasks from the original task network w in a new state s'.

By visualizing the planning problem as a graph, however, the solution seems apparent. We compute the modified task network based on the location of the failure in the task network. Then modify the task network again using the backtrack feature of the planner. And then resume the planning process. During re-planning, the planner marks the nodes that were refined because of this re-planning process.

The task network described in Example 1 is simplistic, and finding the modified task network is trivial. We compute the parent task node of the failed primitive task node and only re-plan for the computed task node. However, for a more complicated task network, this will not work. We will have to come up with a more sophisticated algorithm. Let us understand this with another example.

Example 2. We want to plan for a task network with tasks $t1$, $t2$, and $t3$. Let us assume that the planner generated the solution task network represented in Figure 3(a). We start implementing the primitive tasks in this solution tree as encountered in a DFS tree traversal from the root node. The primitive task sequence or the plan is $\pi = \langle o1, o2, ..., o11, o12\rangle$. However, while executing this plan, assume that $o7$ nondeterministically fails. We need to find the new task network our planner should use for re-planning. Unlike the previous example, replanning just for the parent task node $t4$ of the failed primitive task node $o7$ is incorrect because the failure in executing $o7$ means that $o11$'s preconditions will not be satisfied later in the plan. Thus, additional replanning will be needed in order to prevent the entire plan from failing.

In the above explained scenario, we should modify the solution task network by removing refinements of all the tasks that come after the failed node $o7$ in the Pre-ordered DFS traversal. Alternatively, this could be done more efficiently

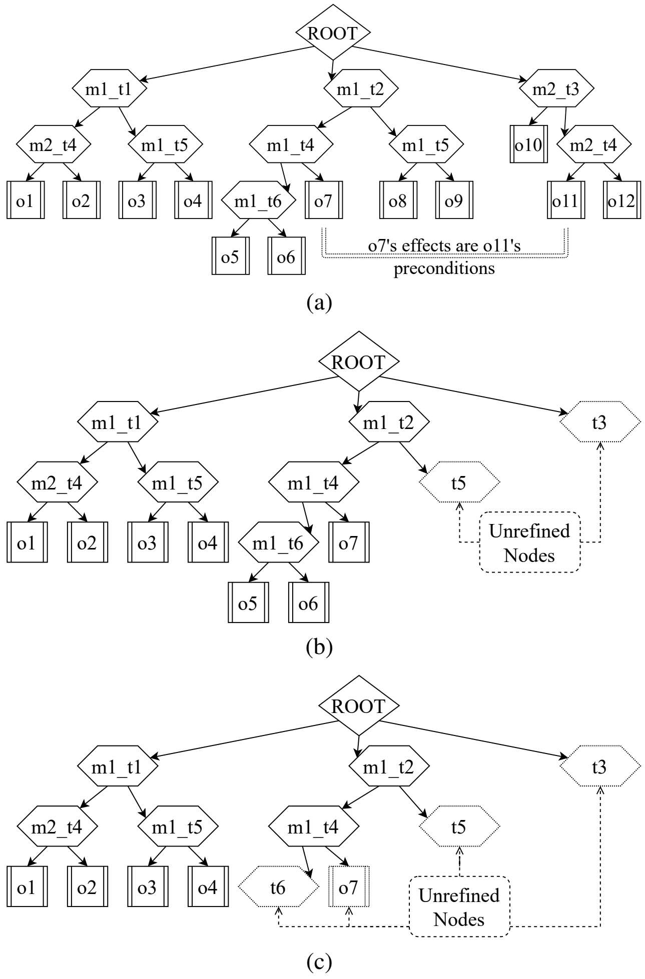


Figure 3: (a) Solution task network after initial planning. (b) Modified task network after failure in execution of $o7$. (c) Modified task network after backtracking from $o7$ on the modified task network in (b). (Example2).

using the Un-Refine-Post algorithm (Algorithm 3). At this point, the modified task network should look like Figure 3(b). Now, we again modify this task network by backtracking on the failed node $o7$. At this point, the modified task network should look like Figure 3(c). We update our model of $o7 \in O$ (if required) used by the planner and perform re-planning again. The planner marks the nodes it refines in this re-planning problem and returns another solution task network for us to execute.

Note that during execution, we only execute the primitive tasks that the planner marked during re-planning. We compute the marked primitive tasks in this solution tree by performing a DFS tree traversal from the root node. ■

This way of repeated planning and acting leads to the formulation of Algorithm 4, Run-Lazy-Refineahead.

Run-Lazy-Refineahead is a repeated planning and acting algorithm for integrating HTN planning and acting. Here, Refineahead is any online HTN planner that provides the solution as a refined task network and provides control over its backtracking feature.

Algorithm 3: Un-Refine-Post. Algorithm used to modify a task network w after failure at u.

```
Un-Refine-Post($w, u$):
while true do
    $p \leftarrow parent(u)$
    foreach $v \in$ BFS_Successors(p) s.t. $v$ after $u$ do
        $refined(v) \leftarrow false$
        if $v$ is non-primitive then
            $W_v \leftarrow descendants(v)$
            $w \leftarrow w \backslash W_v$
    $u \leftarrow p$
    if $u = root(w)$ then
        break
return $w$
```

Algorithm 4: Run-Lazy-Refineahead.

```
Run-Lazy-Refineahead($\Sigma, w$):
$s \leftarrow$ abstraction of observed state $\xi$
while true do
    $w \leftarrow$ Refineahead($\Sigma, s, w$)
    if $w = failure$ then
        return failure
    $\pi \leftarrow$ marked primitive tasks in DFS($w$)
    $a \leftarrow$ first action in $\pi$
    while $\pi \neq \langle\rangle$ and Simulate$(\Sigma, s, \pi) \neq failure$ do
        $a \leftarrow$ pop-first-action($\pi$)
        perform($a$)
        $s \leftarrow$ abstration of observed state $\xi$
    if $\pi \neq \langle\rangle$ then
        $w \leftarrow$ Un-Refine-Post($w, a$)
        $w, a \leftarrow$ Backtrack($w, a$)
    else
        break
```

Run-Lazy-Refineahead executes each plan π as far as possible, calling Refineahead again only when π ends or a plan simulator says that π will no longer work properly. This way of execution can help in environments where it is computationally expensive to call Refineahead, and the actions in π are likely to produce the predicted outcomes. Simulate is the plan simulator, which may use the planner's prediction function γ or may do a more detailed computation (e.g., a physics-based simulation, a Monte-Carlo simulation, et cetera.) that would be too time-consuming for the planner to use. Simulate should return failure if its simulation indicates that π will not work correctly. For example, if it finds that an action in π will have an unsatisfied precondition.

On failure in executing the plan, the tasks refined after the failed task a in the task network w are un-refined using the Un-Refine-Post (Algorithm 3), and backtracking is performed using the Backtrack algorithm of an HTN planner, e.g., Algorithm 2. The resulting task network obtained after these modifications is re-used for the next re-planning process.

Intuitively, deliberative HTN acting implemented in Run-Lazy-Refineahead is more efficient than in Run-Lazy-Lookahead. Since for every re-planning, the Refineahead needs to re-plan only for a subset of the task network, compared to the entire task network for Lookahead, the planning time on average will be lower. Also, since the actions corresponding to the tasks that have already been executed are no longer planned for during re-planning, repetition of already executed tasks will be minimized. Thus, Run-Lazy-Refineahead will lead to executing action sequences with an overall cost less than that by Run-Lazy-Lookahead.

6 Experimental Setup

We used the Robosub Domain (citation removed for blind reviewing) for our experimental evaluation. The Robosub Domain was derived from the RoboSub 2019 competition,[2] where an autonomous underwater vehicle performs various compulsory and optional tasks autonomously to score points in the competition. A planning domain was written for the refinement of these tasks. The planning domain consisted of seventeen primitive task operators and twenty-one task refinement methods for refining ten non-primitive tasks.

We statistically analyzed the performance of Run-Lazy-Lookahead and Run-Lazy-Refineahead approaches for deliberative HTN acting for the above-defined tasks. For the RoboSub competition, we fixed the initial location of the robot and a few other constraints. However, we varied the location of various objects in the planning problem. The initial task network always contains a single task named *competition-task* that needs to be refined to complete all the required tasks based on the competition deliverable.

The planning problem is solved using the IPyHOP planner, and the resulting plan is executed by a simple actor communicating with an execution platform. The execution environment is nondeterministic, which leads to occasional failures in the execution of actions. The repeated planning and acting is done using Run-Lazy-Lookahead and Run-Lazy-Refineahead algorithms. The complete refinement and execution for one such planning problem is termed as a *test case* x, where x_i corresponds to the i^{th} planning problem with the initial state s_i, where s_i is the i^{th} state in I. We repeat this deliberative HTN acting process for all initial states. The execution of all the test cases x_i is known as an *experiment*, e, where $e = \langle x_1, x_2, ..., x_j \rangle$, where $j = \|I\|$. We repeat the experiment 11 times, i.e. $E = \langle e_1, e_2, ..., e_{11} \rangle$.

We evaluate performance with three metrics:

- **Total iterations taken**: This metric calculates the total number of iterations taken by the planner for a given test case. Calculating iterations provides a good estimate of the planner's total planning time for a test case.
- **Total action cost**: This measures the total cost of an action sequence for a given test case. Execution of smaller action sequences will generally lead to lower total action costs.
- **Final state reward**: This measures the reward obtained based on the final state of the robot in a test case. This is a good indicator of how well the competition task was completed.

The raw data collected d_{raw} in each experiment $e \in E$ described earlier was accumulated into a single dataset D_{raw}, where $D_{raw} = \langle d_1, d_2, ..., d_{11} \rangle$. D_{raw} was post-processed to calculate the required metrics and the results were stored in a single numpy array representing the results dataset $D_{results}$. Let the size of the dataset $D_{results}$ be $[\|e\| \times \|a\| \times \|x\| \times \|m\|]$. Here $\|e\| = 11$ is the number of experiments performed, $\|a\| = 2$ is the number of deliberative HTN acting algorithms being compared, $\|x\| = 10000$ is the number of test cases solved, and $\|m\| = 3$ is the number of metrics evaluated.

The $D_{results}$ dataset was processed further by doing a reduce mean operation across the $zero^{th}$ axis of the dataset. Thus the dataset D_{exp_mean} of size $[\|a\| \times \|x\| \times \|m\|]$ was generated. Each element in the dataset D_{exp_mean} represents the mean value of a metric for a given test case across experiments. Since the value of a metric for a given test case varies across experiments due to the non-determinism of the execution environment, taking the mean across experiments gives us a more reliable estimate of that metric for a given test case. The metrics calculated in the dataset D_{exp_mean} are illustrated in Figures 4(a), 4(b), and 4(c).

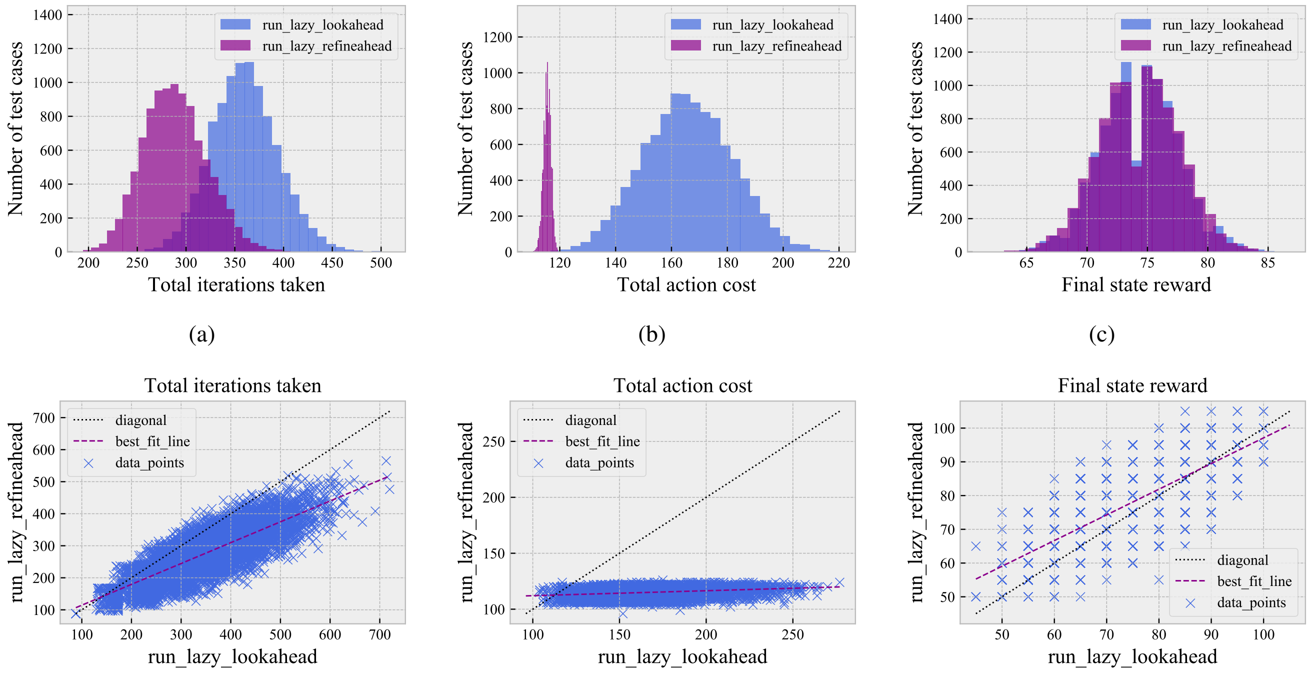
It is possible that different numbers of failures occur with each test case and a more fair assessment would compare only situations with the same number of failures. Another form of post-processing was done on D_{raw} to generate the D_{eqv} dataset to balance this concern and these results are presented in Figures 4(d), 4(e), and 4(f). The values represented by the D_{eqv} dataset are less accurate since they only use a single data point for a metric of a given test case. Comparatively, the metric measurements from D_{exp_mean} are computed by performing a mean operation across 11 values for each metric in a test case. To improve the accuracy of the metric measurements by D_{eqv} dataset, we will need to perform more experiments such that multiple data points are available for each metric in the dataset.

7 Results and Discussion

Our results suggest that Run-Lazy-Refineahead is a better algorithm for deliberative HTN acting compared to Run-Lazy-Lookahead. In Figure 4(a), we show the values of the metric - the total number of iterations taken by the planner, for Run-Lazy-Lookahead (blue histogram) and Run-Lazy-Refineahead (purple histogram). The relation of this metric for the two deliberative acting algorithms is visualized in Figure 4(d) as a scatter plot. Based on our results, we can state that the Run-Lazy-Refineahead leads to the generation of shorter and easily solvable re-planning problems. Also, we can see in Table 2 that the average time spent in planning during Run-Lazy-Refineahead is $\approx 80\%$ of the average time spent in planning during Run-Lazy-Lookahead.

Figure 4(b) shows the values of the total-action-cost metric, for Run-Lazy-Lookahead and Run-Lazy-Refineahead. The relation of this metric for the two deliberative acting al-

[2] https://robosub.org/programs/2019/



(a)

(b)

(c)

(d)

(e)

(f)

Figure 4: Results of three metrics: total iterations (left), action cost (middle), and final state reward (right). Each pair shows the distribution visualized using histograms (top, using D_{exp_mean}) and the relation visualized by fitting a line on the scatter plot (bottom, using D_{eqv}).

Metric	Run-Lazy-Lookahead		Run-Lazy-Refineahead	
	Mean	SD	Mean	SD
Total iterations taken	364.470	34.178	290.592	32.639
Total action cost	165.928	16.002	115.478	1.339
Final State Reward	74.368	3.183	74.326	3.242

Table 1: Overview of results obtained using D_{mean_exp}

Metric	Refineahead / Lookahead Mean	Best-fit line	
		Slope	Y-intercept
Total iterations taken	0.796	0.639	58.296
Total action cost	0.682	0.044	108.282
Final State Reward	1.049	0.805	14.540

Table 2: Overview of results obtained using D_{eqv}

gorithms is visualized in Figure 4(e). Thus, we can state that the Run-Lazy-Refineahead leads to the execution of smaller action sequences. In Table 2 we can see that the average cost of executing action sequences generated from Run-Lazy-Refineahead is $\approx 70\%$ of the average cost of executing action sequences generated from Run-Lazy-Lookahead.

Figure 4(c) shows show the values of the final-state-reward metric, for Run-Lazy-Lookahead and Run-Lazy-Refineahead. The relation of this metric for the two deliberative acting algorithms is shown in Figure 4(f). The results show that the improvements mentioned earlier were realized without sacrificing the average final state reward.

There is also a hidden burden associated with using the Run-Lazy-Lookahead algorithm not portrayed by our experiments. Authoring the domain for use in the Run-Lazy-Lookahead algorithm requires accounting for numerous scenarios where failures would lead to repeated tasks, getting stuck in infinite task loops, getting stuck in non-recoverable states, et cetera. These problems can be addressed by clever definitions of task methods and flags in the state. However, it might not be possible to eliminate these undesirable behaviors. In more modest domain model definitions like ours, this problem is not as pronounced. However, as the domain models get more and more comprehensive, this problem quickly worsens. In Run-Lazy-Refineahead, however, the planner always resumes after backtracking on the node that caused the failure. Thus, repetition of tasks and other unexpected behaviors are minimized.

For our experiments, every effort was made to make deliberative HTN acting using Run-Lazy-Lookahead as efficient as possible. Optimizing the performance of the Run-Lazy-Lookahead algorithm was our prime focus. The task methods, operators, and state definition were designed primarily for use in the Run-Lazy-Lookahead algorithm. Then the same domain model definition and state definition were used for the Run-Lazy-Refineahead algorithm. This reuse of domain definition leads to the planner performing many unnecessary constraint checks during task refinement required for Run-Lazy-Lookahead but are not required for Run-Lazy-Refineahead. The domain authoring for use in

Run-Lazy-Refineahead is much more straightforward and concise. If the domain model definition was primarily designed for Run-Lazy-Refineahead, the results would considerably shift in its favor. The metrics would remain the same for Run-Lazy-Refineahead but significantly worsen for the Run-Lazy-Lookahead. However, even though the calculated metrics would remain the same, the second execution would be computationally faster than the first since simpler domain model definitions are being used for task refinement process.

Hence we can comfortably state that Run-Lazy-Refineahead is a better alternative to Run-Lazy-Lookahead for deliberative HTN acting.

8 Summary and Future work

In this paper we have presented new algorithms for integrated HTN planning and acting.

The first main contribution is an HTN planner, IPyHOP. IPyHOP is an iterative tree traversal-based HTN planning algorithm written in Python that provides extensive control over its task network refinement. Since the algorithm is iteration-based, the task network refinement can be paused, modified, and resumed at the user's discretion. This level of control makes it a great choice for planning in scenarios where re-planning is required. Since IPyHOP uses the Python programming language, authoring domain model definitions does not require developers to learn specialized programming languages. Instead, developers can write the task methods as Python functions. Also, since it follows an object-oriented design, it is effortless to integrate and debug it with other computer programs. IPyHOP is envisioned to make HTN planning accessible to a much broader audience who were earlier reluctant to adopt it for their planning problems due to a lack of HTN planners in Python.

The second main contribution is a deliberative HTN actor, Run-Lazy-Refineahead. Run-Lazy-Refineahead is a repeated planning and acting algorithm specially designed for deliberative HTN acting. We showed experimentally that it performs better for deliberative HTN acting than Run-Lazy-Lookahead, a popular acting-and-planning algorithm. Run-Lazy-Refineahead uses the hierarchical nature of the refined task network generated by HTN planners like IPyHOP to develop smaller and smaller task refinement problems as the execution proceeds. The improvement can be beneficial in deliberative HTN acting in fast-moving dynamic worlds like in games or in robotics scenarios.

We hope that the large community of roboticists and game developers who program their systems in Python adopt IPyHOP, and Run-Lazy-Refineahead for HTN planning, and integrated planning and acting.

8.1 Limitations and Future Work

In some aspects, HTN planning is quite controversial. The controversy lies in its requirement for well-conceived and well-structured domain knowledge. Such knowledge is likely to contain rich information and guidance on how to solve a planning problem, thus encoding more of the solution than was envisioned for classical planning systems. This structured and rich knowledge gives a primary advantage to HTN planners in terms of speed and scalability when applied to real-world problems compared to their counterparts in the classical planning world. However, this also makes their performance depend on the users' definition of suitable domain-specific task methods.

IPyHOP faces many of the same challenges as other HTN planners, namely:

- Domain engineering effort in writing methods: The HTN formalism requires implementing methods to cover every possible scenario that the agent could encounter. An HTN planner trying to plan for an unanticipated state may fail without returning a solution.
- Brittleness in open and dynamic environments: The previous problem is intensified in open, dynamic environments. Nondeterministic events or outcomes can result in unanticipated situations, and HTN planners are not well suited to work in open and dynamic environments.
- Effective domain-independent HTN planning heuristics: Heuristics are crucial in guiding an algorithm toward high-quality solutions. HTN planners often rely heavily on the user-provided knowledge through the definition of methods in providing the necessary guidance.

These limitations are important areas for future research on improving IPyHOP.

In some aspects, the integration of HTN planning and acting using Run-Lazy-Refineahead that we proposed here can be interpreted as a simple HTN planner *guided* acting. Some algorithms directly integrate a planner's descriptive model into a hierarchical actor to select refinement methods, while others directly integrate planners that plan using operational representations with the actor RAE e.g., (Patra et al. 2019, 2020). Combining a hierarchical planner and an actor using this strategy leads to much more efficient and tighter integration. We believe a similar form of integration is also possible for HTN planners and HTN actors. An HTN planner like IPyHOP could be directly integrated with an HTN actor like RAE-lite, where the HTN actor would decide on the method it uses for task refinement based on recommendation of the HTN planner.

For hierarchical acting and planning, there are two main ways to represent an objective: tasks and goals. A task is an activity to be accomplished by an actor, while a goal is a final state that should be reached. Depending on a domain's properties and requirements, users can choose between task-based and goal-based approaches. Since IPyHOP is based on GTPyhop (Nau et al. 2021), it supports both HTN and HGN planning. However, we have not made any use of HGN planning in this paper. For future work, we intend to do experimental evaluations of Run-Lazy-Refineahead versus Run-Lazy-Lookahead on HGN versions of our test domains.

9 Acknowledgments

This work has been supported in part by ONR grant N000142012257 and NRL grants N0017320P0399 and N00173191G001. The information in this paper does not necessarily reflect the position or policy of the funders, and no official endorsement should be inferred.

References

Bansod, Y. 2021. *Refinement Acting vs. Simple Execution Guided by Hierarchical Planning*. Master's thesis, University of Maryland. URL https://www.cs.umd.edu/users/nau/others-papers/bansod2021refinement.pdf.

Bauters, K.; Liu, W.; Hong, J.; Sierra, C.; and Godo, L. 2014. Can(Plan)+: Extending the operational semantics of the BDI architecture to deal with uncertain information. In *UAI*.

Castillo, L.; Fdez-Olivares, J.; García-Pérez, Ó.; and Palao, F. 2005. Temporal enhancements of an HTN planner. In *Conf. Spanish Assoc. for Artificial Intelligence*, 429–438.

Chen, D.; and Bercher, P. 2021. Fully observable nondeterministic HTN planning – formalisation and complexity results. In *ICAPS*, volume 31, 74–84.

Clement, B. J.; Durfee, E. H.; and Barrett, A. C. 2007. Abstract reasoning for planning and coordination. *Journal of Artificial Intelligence Research* 28: 453–515.

Currie, K.; and Tate, A. 1991. O-Plan: the open planning architecture. *Artificial intelligence* 52(1): 49–86.

De Silva, L.; and Padgham, L. 2005. Planning on demand in BDI systems. In *ICAPS (Poster)*.

Erol, K. 1996. *Hierarchical task network planning: formalization, analysis, and implementation*. Ph.D. thesis, University of Maryland.

Feldman, Z.; and Domshlak, C. 2013. Monte-Carlo planning: Theoretically fast convergence meets practical efficiency. *arXiv preprint arXiv:1309.6828* .

Feldman, Z.; and Domshlak, C. 2014. Monte-Carlo tree search: To MC or to DP? In *ECAI*, 321–326.

Ghallab, M.; Nau, D.; and Traverso, P. 2016. *Automated Planning and Acting*. Cambridge University Press.

Goldman, R. P.; and Kuter, U. 2019. Hierarchical task network planning in Common Lisp: the case of SHOP3. In *Proc. European Lisp Symposium*, 73–80.

Hogg, C.; Kuter, U.; and Muñoz-Avila, H. 2009. Learning hierarchical task networks for nondeterministic planning domains. In *IJCAI*, 1708–1714.

Ingrand, F.; and Ghallab, M. 2017. Deliberation for autonomous robots: a survey. *Artificial Intelligence* 247: 10–44.

Kuter, U.; and Nau, D. S. 2005. Using domain-configurable search control for probabilistic planning. In *AAAI*, 1169–1174.

Menif, A.; Jacopin, É.; and Cazenave, T. 2014. SHPE: HTN planning for video games. In *Workshop on Computer Games*, 119–132. Springer.

Musliner, D.; Pelican, M. J.; Goldman, R. P.; Kresbach, K. D.; and Durfee, E. H. 2008. The evolution of CIRCA, a theory-based AI architecture with real-time performance guarantees. In *AAAI Spring Symp.: Emotion, Personality, and Social Behavior*.

Nau, D. 2013a. Game applications of HTN planning with state variables. In *ICAPS Workshop on Planning in Games*.

Nau, D.; Cao, Y.; Lotem, A.; and Munoz-Avila, H. 1999. SHOP: Simple hierarchical ordered planner. In *Proc. 16th IJCAI*, 968–973.

Nau, D.; Patra, S.; Roberts, M.; Bansod, Y.; and Li, R. 2021. GTPyhop: A hierarchical goal+task planner implemented in Python. In *ICAPS Workshop on Hierarchical Planning (HPlan)*.

Nau, D. S. 2013b. Pyhop, version 1.2.2: A simple HTN planning system written in Python. Software release. URL https://bitbucket.org/dananau/pyhop/src/master/.

Nau, D. S.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN planning system. *JAIR* 20: 379–404.

Neufeld, X.; Mostaghim, S.; Sancho-Pradel, D. L.; and Brand, S. 2017. Building a planner: A survey of planning systems used in commercial video games. *IEEE Transactions on Games* 11(2): 91–108.

Patra, S.; Ghallab, M.; Nau, D.; and Traverso, P. 2019. Acting and planning using operational models. In *AAAI*, 7691–7698.

Patra, S.; Mason, J.; Kumar, A.; Ghallab, M.; Traverso, P.; and Nau, D. 2020. Integrating acting, planning, and learning in hierarchical operational models. In *ICAPS*, 478–487.

Pollack, M. E.; and Horty, J. F. 1999. There's more to life than making plans: plan management in dynamic, multiagent environments. *AI Magazine* 20(4): 71–71.

Sacerdoti, E. 1975. The Nonlinear Nature of Plans. In *IJCAI*, 206–214.

Sardina, S.; De Silva, L.; and Padgham, L. 2006. Hierarchical planning in BDI agent programming languages: A formal approach. In *Proc. 5th AAMAS*, 1001–1008.

Shivashankar, V.; Kuter, U.; Nau, D. S.; and Alford, R. 2012. A hierarchical goal-based formalism and algorithm for single-agent planning. In *AAMAS*, 981–988.

Tate, A. 1977. Generating project networks. In *Proc. 5th IJCAI*, 888–893.

Tate, A.; Drabble, B.; and Kirby, R. 1994. O-Plan2: an open architecture for command, planning and control. In Zweben, M.; and Fox, M. S., eds., *Intelligent Scheduling*. Morgan Kaufmann.

Teichteil-Koenigsbuch, F.; Infantes, G.; and Kuter, U. 2008. RFF: A robust, FF-based MDP planning algorithm for generating policies with low probability of failure. In *Sixth International Planning Competition at ICAPS*.

Wilkins, D. E. 1990. Can AI planners solve practical problems? *Computational intelligence* 6(4): 232–246.

Yao, Y.; Alechina, N.; Logan, B.; and Thangarajah, J. 2021. Intention progression using quantitative summary information. In *Proc. 20th AAMAS*, 1416–1424.

Yoon, S. W.; Fern, A.; and Givan, R. 2007. FF-Replan: a baseline for probabilistic planning. In *ICAPS*, 352–359.

Yoon, S. W.; Fern, A.; Givan, R.; and Kambhampati, S. 2008. Probabilistic planning via determinization in hindsight. In *AAAI*, 1010–1016.