HTN Replanning from the Middle

Yash Bansod[1], **Sunandita Patra**[2, *], **Dana Nau**[1,2], **Mark Roberts**[3]
[1]Institute for Systems Research and [2]Dept. of Computer Science, Univ. of Maryland, College Park, MD, USA
[3]Navy Center for Applied Research in AI, Naval Research Laboratory, Washington, DC, USA
*now at JP Morgan AI Research, New York, USA
{yashb, patras, nau}@umd.edu, {mark.roberts}@nrl.navy.mil

Abstract

When an actor executes a plan, action failures and exogenous events may lead to unexpected states that require replanning from the middle of plan execution. In Hierarchical Task Network (HTN) planning, unless the HTN methods have been carefully written to work well in unexpected states, replanning may either fail or produce plans that perform poorly.

To overcome this problem, we introduce IPyHOP, a reentrant version of GTPyhop (a SHOP-like HTN planner), and Run-Lazy-Refineahead, a modification of the Run-Lazy-Lookahead actor that utilizes IPyHOP's reentrant replanning capability to replan during plan execution. In our experiments, Run-Lazy-Refineahead and IPyHOP expend less search effort (fewer decompositions and fewer iterations), find revised plans with fewer actions and lower total action cost, and finish execution with fewer failures.

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 the execution of the plan may fail because of (i) failure in the execution of actions, (ii) occurrence of unexpected events, or (iii) if 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; Patra *et al.* 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 has limited their adoption, but they have been highly influential. Algorithm 1 gives a version of SHOP's pseudocode.

Python is a much more widely adopted programming language used by roboticists, game developers, ML engineers, and AI engineers. The pyhop planner (Nau 2013a; Nau 2013b) 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. To resolve this problem, we introduce a new hierarchical planner that allows an actor to plan from the middle of its current state *and* current plan by reusing the plan and decomposition tree to resume planning from the point of the plan failure. Our primary contributions are:

1. We describe IPyHOP[2], 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. This way of re-entrant planning for partially solved HTNs is unique to IPyHOP. An important part of this adaptation is to maintain the history of the solution tree to allow backtracking, which is necessary for replanning during acting.
2. Inspired by the RAE actor in (Ghallab *et al.* 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 net-

[1]There are some exceptions, e.g., (Kuter and Nau 2005; Hogg *et al.* 2009; Chen and Bercher 2021).

[2]https://github.com/YashBansod/IPyHOP

Algorithm 1: The planning algorithm used in SHOP, pyhop, and the HTN-planning part of GTPyhop.

```
1  SHOP (state $s$, task-list $T$)
2    if $T =$ nil then return nil
3    $t \leftarrow$ the first task in $T$
4    $U \leftarrow$ the remaining tasks in $T$
5    if $t$ is primitive and there is an operator $o$
6          that matches $t$ and is applicable in $s$ then
7       $s \leftarrow$ the state produced by applying $o$ to $s$
8       $\pi \leftarrow$ SHOP$(s, U)$
9       if $\pi =$ fail then return fail
10      return $o + \pi$
11   else if $t$ is non-primitive and there is a method
12         $m$ that is relevant for $t$ and applicable in $s$ then
13      return SHOP$(s,$ subtasks$(m), U))$
14   else return fail
```

work 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.

2 Background: pyhop and GTPyhop

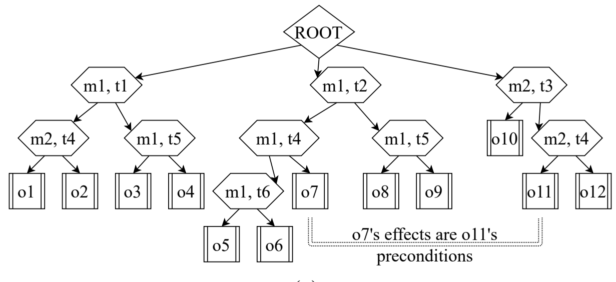
GTPyhop (Nau *et al.* 2021) is a domain-independent Goal Task Network (GTN) planning system written in Python. GTPyhop is a progressive totally-ordered GTN planner that combines the task- and goal-decomposition strategies used in SHOP (Nau *et al.* 1999) and GDP (Shivashankar *et al.* 2012). It plans for a sequence of tasks and goals in the same order that they will later be executed, using recursion for task and goal refinement. We only use the HTN portion of GTPyhop (Algorithm 1) in these studies and leave extensions to goal networks for future work. We use the terms refine and decompose interchangeably in this paper.

Algorithm 1's search space is an AND/OR tree. The initial task list T is an AND-branch, the set of method instances that match each task t are an OR-branch, the subtasks of each method instance are an AND-branch, and each operator instance is a leaf node. The algorithm looks for a *solution tree* (see Figure 1(a) for an example) that contains one method instance at each OR branch for all of the tasks at each AND-branch. The returned plan π is the sequence of operator instances at the leaf nodes of the solution tree.

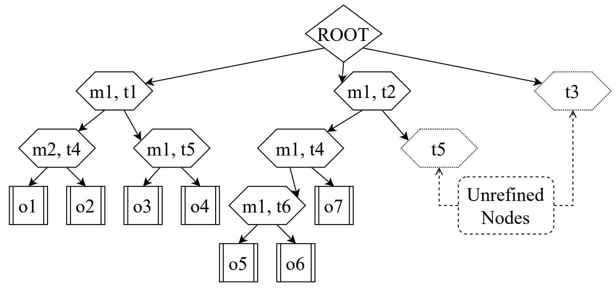
3 HTN Planning in IPyHOP

Like most HTN and HGN planners, GTPyhop has two limitations that impact 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. Second, GTPyhop returns a plan π, but not the solution tree that produced π. To replan after an action failure, the planner needs to know not only what action failed, but what tasks it was trying to achieve at that point in the plan. This requires a copy of the solution tree.

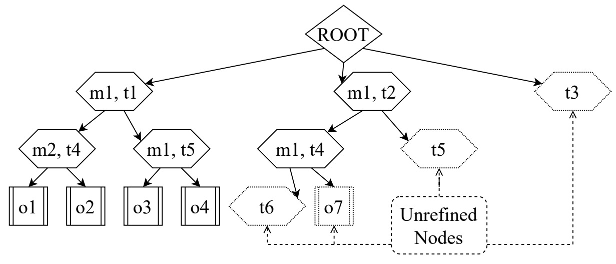
For example, consider Figure 1(a) which shows a notional hierarchical plan for tasks $t1$, $t2$, and $t3$, where operator instances (i.e., actions) begin with o, and where decomposition



(a)



(b)



(c)

Figure 1: HTN solution trees to illustrate the challenge of replanning during plan execution. Part (a) is after initial planning, part (b) is after the execution failure of $o7$, and part (c) is after backtracking from $o7$.

methods begin with m. The root node represents the initial task list, each hexagonal node shows a method instance m_i used for a task t_j, and each rectangular node represents an operator instance o_i. The expected sequence of actions, the plan, $\pi = \langle o1, o2, ..., o11, o12 \rangle$ is produced using a Depth First Search (DFS) tree traversal; note that $o7$ produces effects on which $o11$ relies. Suppose while executing π, $o7$ nondeterministically fails. Replanning for only the parent task node of $t4$ (that is, $t2$) to resolve the failure of $o7$ may not be correct because $o11$'s preconditions may not be satisfied. Instead, replanning must start by removing refinements for all tasks executing after the failed node $o7$, resulting in the network of Figure 1(b), and then backtracking from $o7$ to produce Figure 1(c), from which replanning can begin. Accomplishing this transformation is non-trivial and requires several extensions to HTN planning and acting.

To overcome the above limitations, IPyHOP uses an iterative tree traversal procedure for task decomposition and backtracking and returns the entire solution tree rather than

Algorithm 2: IPyHOP's HTN pseudocode.

```
IPyHOP(current state $s$, decomposition tree $w$):
    $p \leftarrow w$'s root
    loop
        if all tasks in $w$ have been expanded then return $w$
        $u \leftarrow$ the first un-expanded node of $w$
        if $u$ has been visited then
            $s \leftarrow$ state($u$)        # use the cached state
        else state($u$) $\leftarrow s$      # cache the current state
        mark $u$ as visited
        $t \leftarrow$ task($u$)
        if $t$ matches an operator $o$ then
            if $o$ is applicable in $s$ then
                $s \leftarrow$ the state produced by applying $o$ to $s$
                mark $u$ as expanded
        else
            for each method $m$ that matches $t$
                if $m$ hasn't been already tried for $t$
                    if $m$ is applicable in $s$ then
                        mark $u$ as expanded
                        install $m$'s subtasks as children of $u$
                        exit the for loop
        if $u$ hasn't been expanded then
            backtrack($w, u$)

backtrack($w, u$):
    $v \leftarrow$ non-primitive task node expanded before $u$
    un-expand all nodes expanded after and including $v$
```

just the solution plan. This allows IPyHOP to accept a partial solution tree with a failed action marked as a backtracking point (see Figure 1(b)), so that it can search for a different solution tree (see Figure 1(c)). IPyHOP implements both HTN and HGN planning and replanning, but we focus only on the HTN part of IPyHOP.

Algorithm 2 shows the IPyHOP algorithm, accepting the current state s, and a (partial) solution tree w. At the start of iteration (Lines 4, 5) we determine if there are any unexpanded nodes. If all nodes are expanded, the algorithm returns the solution tree. Otherwise, it proceeds with expanding the next node. This can be naively obtained using pre-ordered DFS on w. However, repeated pre-ordered DFS on w to find un-expanded nodes is inefficient. We do this more efficiently using some pointer manipulations. If the node has been visited, then we re-use the cached state at that node, otherwise, the current state is cached into the node (Lines 6–8). The node can be expanded in two ways. Either the task t corresponding to node u is a primitive task that matches an operator o. And, applying the operator o at current state s leads to a valid new state s (Lines 11–14). Or if t is a non-primitive task and there exists a task-method m that hasn't already been used to decompose it into simpler tasks (Lines 15–21). If the node u could not be expanded, then the algorithm backtracks the decomposition (Lines 22, 23).

The **backtrack** procedure accepts the partial solution w and the current node u from which backtracking should proceed. It first finds the non-primitive task node v expanded just before u. This is done by searching the descendants of u's parent. Then it un-expands all nodes expanded after and including v.

Algorithm 3: Run-Lazy-Refineahead, where underlines indicate changes from Run-Lazy-Lookahead.

```
Run-Lazy-Refineahead($\Sigma$,$\underline{w}$):
    $s \leftarrow$ abstraction of observed state $\xi$; $\underline{f \leftarrow \phi}$
    $\underline{\text{loop}}$
        $\underline{w \leftarrow \text{Refineahead}(\Sigma, s, w, f)}$
        $\underline{\text{if } w = \textit{failure} \text{ then return } \textit{failure}}$
        $\pi \leftarrow$ marked primitive tasks in DFS($w$)
        while $\pi \neq \langle\rangle$ and Simulate($\Sigma, s, \pi$) $\neq$ fail do
                $a \leftarrow$ pop-first-action($\pi$); perform($a$)
                $s \leftarrow$ abstration of observed state $\xi$
        $\underline{f \leftarrow \text{action that leads to failure}}$
```

4 Integrating IPyHOP with an Actor

An actor is a piece of software that executes plans to complete a goal or task. We integrate IPyhop into an acting engine that extends the Run-Lazy-Lookahead algorithm introduced by Ghallab, Nau, & Traverso (Chapter 2 2016).

Algorithm 3 shows Run-Lazy-Refineahead, which modifies Run-Lazy-Lookahead (changes from the prior algorithm are underlined) by (1) in Line 1, changing the input from a goal to a partial HTN w; (2) in Line 2, keeping a track of failed action explicitly; (3) in Line 3, changing the loop condition from the goal being entailed in the current state to an infinite loop; (4) in Line 4, calling a planner that accepts the partial solution w and a failed action f, which in this case is IPyHOP; and (5) in Line 10 updating the pointer to the action node that causes execution failure.

The main difference between this acting algorithm and Run-Lazy-Lookahead is the use of a partial solution w and the call to the IPyHOP planner, which can plan from the middle of execution failure. Note that when IPyHOP is used for replanning, it first updates the cached state in all its expanded nodes to the newly provided state and then backtracks from the provided failure node. Here, **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) that would be too time-consuming for the planner to use.

5 Experiments

We expect Run-Lazy-Refineahead to be more efficient than Run-Lazy-Lookahead because it only replans a subset of the task network and will not repeat action sequences for tasks already completed. To assess this, we evaluated the performance of Run-Lazy-Lookahead (with GTPyhop) and Run-Lazy-Refineahead (with IPyHOP) in two HTN planning and acting domains: RoboSub and Rescue described below. Both execution environments are nondeterministic, which leads to occasional failures in action executions. We evaluate performance using two metrics. **Total decompositions** measures how many nodes were expanded during a search. **Total action cost** measures the total cost of an action sequence for a given test.

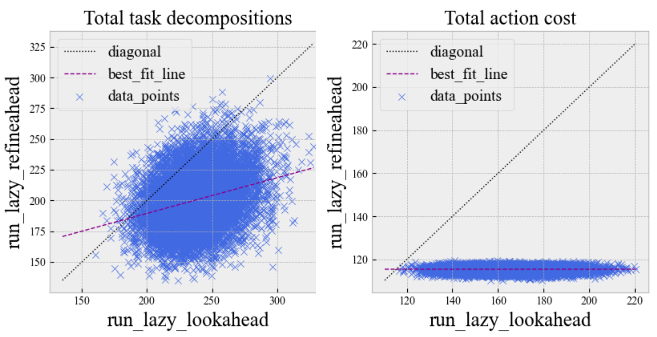


Figure 2: RoboSub decompositions (left) and action cost (right).

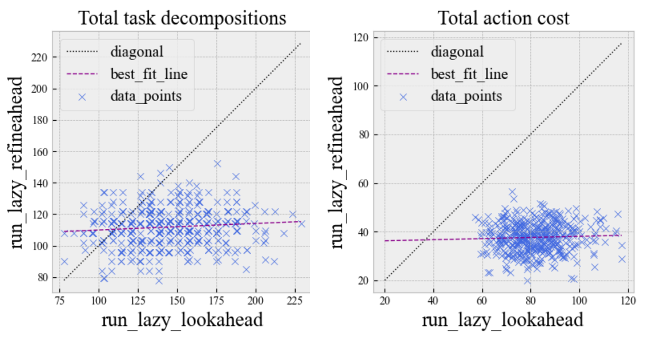


Figure 3: Rescue decompositions (left) and action cost (right).

5.1 Results: RoboSub Domain

The Robosub Domain (Bansod 2021) is derived from the RoboSub 2019 competition, where an autonomous underwater vehicle performs various compulsory and optional tasks autonomously to score points in the competition. We wrote a planning domain for the refinement of these tasks that consists of seventeen primitive task operators and twenty-one task refinement methods for refining ten non-primitive tasks.

For this evaluation, 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 and sampled 1000 random starting states. For each of the 1000 starting states, we perform deliberative acting using Run-Lazy-Lookahead and Run-Lazy-Refineahead 11 times and collect the performance data. 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.

Figure 2 shows scatterplots for two metrics on the *mean* data. Run-Lazy-Refineahead generally decomposes fewer nodes (left) and produces executions with substantially lower total action cost (right). It also produced execution runs with similar final rewards (not shown due to limited space). These results suggest that Run-Lazy-Refineahead is a better algorithm than Run-Lazy-Lookahead because it is more efficient at producing lower-cost plans that achieve a similar final reward.

5.2 Results: Rescue Domain

The Rescue Domain is derived from (Patra *et al.* 2021), where a team of autonomous agents (comprised of UAVs, and UGVs) collaboratively survey and rescue victims from a map after a calamity. We wrote a planning domain for the refinement of these tasks that consists of twenty primitive task operators and twenty-four task refinement methods for refining ten non-primitive tasks.

For this evaluation, we fixed the initial location of the robots, the amount of medicine they carry, and a few other constraints. However, we varied the location of obstacles, injured humans, and demolished locations. Similar to RoboSub, we generated 1000 starting states for the execution and performed 11 runs of each point. One distinguishing detail of the Rescue domain is that the agent cannot skip tasks during planning or execution. This can result in unrecoverable planning failures. Of the 1000 problems in consideration, 39 were unsolvable using our domain definition. Of the remaining 961 problems, Run-Lazy-Refineahead failed for 29 problems, whereas Run-Lazy-Lookahead failed for 493 problems. Already, it is clear that Run-Lazy-Refineahead failed substantially fewer times.

Figure 3 shows the results for this domain on cases where at least one algorithm succeeded on every problem, which occurred for 459 of the 1000 starting points. Similar to the results on RoboSub, Run-Lazy-Refineahead expanded fewer nodes (left) and produces executions that result in substantially lower action costs (right).

5.3 Discussion

We can comfortably state that Run-Lazy-Refineahead is better than Run-Lazy-Lookahead for deliberative HTN acting.

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 not 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 the task refinement process.

6 Related Work

HTN and HGN planning. One of the first HTN planners was NOAH (Sacerdoti 1975). Numerous HTN planners have been developed since then. Some of the best-known ones are Nonlin (Tate 1977), SIPE and SIPE-2 (Wilkins 1990), O-Plan (Currie and Tate 1991) and O-Plan2 (Tate *et al.* 1994), UMCP (Erol 1996), SHOP, SHOP2, and SHOP3 (Nau *et al.* 1999; Nau *et al.* 2003; Goldman and Kuter 2019), and SIADEX (Castillo *et al.* 2005). Among other applications, HTN planning is widely used in the gaming industry (Neufeld *et al.* 2017). Some HTN planners, e.g., Simple Hierarchical Planning Engine (SHPE) (Menif *et al.* 2014) are specifically designed for AI planning in video games.

HGN planners, e.g., GDP and GoDeL (Shivashankar *et al.* 2012; Shivashankar *et al.* 2013), are like HTN planners except that they decompose goals instead of tasks. The task-and-goal decomposition strategy used in GTPyhop and IPyHOP combines aspects of SHOP and GDP.

Planning and acting. In the CIRCA system (Musliner *et al.* 2008), the current 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. The Run-Lookahead and Run-Lazy-Lookahead in (Ghallab *et al.* 2016, Chapter 2) use a somewhat similar strategy. Each time Run-Lookahead calls its planner, it performs only the first action of the returned plan, then calls the planner again. Run-Lazy-Lookahead executes each plan as far as possible, calling the planner 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 *et al.* 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.

7 Summary and Future work

We presented new algorithms for integrated HTN planning and acting. We introduced IPyHOP, an iterative tree traversal-based HTN planning algorithm written in Python that provides extensive control over its task network refinement. We also introduced Run-Lazy-Refineahead, a repeated planning and acting algorithm specially designed for deliberative HTN acting. 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 robotics scenarios. We showed experimentally that it performs better for deliberative HTN acting than Run-Lazy-Lookahead. 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.

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; Patra *et al.* 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 the 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.

Acknowledgments.

This work has been supported for UMD in part by ONR grant N000142012257 and NRL grants N0017320P0399 and N00173191G001. MR thanks ONR and NRL for funding this research. The information in this paper does not necessarily reflect the position or policy of the funders, and no official endorsement should be inferred. [3]

[3]**Disclaimer** This paper was prepared for informational purposes in part by the Artificial Intelligence Research group of JPMorgan Chase & Co. and its affiliates ("JP Morgan"), and is not a product of the Research Department of JP Morgan. JP Morgan makes no representation and warranty whatsoever and disclaims all liability, for the completeness, accuracy or reliability of the information contained herein. This document is not intended as investment research or investment advice, or a recommendation, offer or solicitation for the purchase or sale of any security, financial instrument, financial product or service, or to be used in any way for evaluating the merits of participating in any transaction, and shall not constitute a solicitation under any jurisdiction or to any person, if such solicitation under such jurisdiction or to such person would be unlawful.

References

[Bansod 2021] Yash Bansod. Refinement acting vs. simple execution guided by hierarchical planning. Master's thesis, University of Maryland, 2021.

[Bauters *et al.* 2014] Kim Bauters, Weiru Liu, Jun Hong, Carles Sierra, and Lluis Godo. Can(plan)+: Extending the operational semantics of the BDI architecture to deal with uncertain information. In *UAI*, 2014.

[Castillo *et al.* 2005] Luis Castillo, Juan Fdez-Olivares, Óscar García-Pérez, and Francisco Palao. Temporal enhancements of an HTN planner. In *Conf. Spanish Assoc. for Artificial Intelligence*, pages 429–438, 2005.

[Chen and Bercher 2021] Dillon Chen and Pascal Bercher. Fully observable nondeterministic htn planning – formalisation and complexity results. In *ICAPS*, pages 74–84, 2021.

[Currie and Tate 1991] Ken Currie and Austin Tate. O-Plan: the open planning architecture. *Artificial intelligence*, 52(1):49–86, 1991.

[De Silva and Padgham 2005] Lavindra De Silva and Lin Padgham. Planning on demand in BDI systems. In *ICAPS (Poster)*, 2005.

[Erol 1996] Kutluhan Erol. *Hierarchical task network planning: formalization, analysis, and implementation*. PhD thesis, University of Maryland, 1996.

[Ghallab *et al.* 2016] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning and Acting*. Cambridge University Press, 2016.

[Goldman and Kuter 2019] Robert P Goldman and Ugur Kuter. Hierarchical task network planning in common lisp: the case of SHOP3. In *Proc. European Lisp Symp.*, pages 73–80, 2019.

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

[Ingrand and Ghallab 2017] Félix Ingrand and Malik Ghallab. Deliberation for autonomous robots: a survey. *Artificial Intelligence*, 247:10–44, 2017.

[Kuter and Nau 2005] Ugur Kuter and Dana Nau. Using domain-configurable search control for probabilistic planning. In *AAAI*, pages 1169–1174, July 2005.

[Menif *et al.* 2014] Alexandre Menif, Éric Jacopin, and Tristan Cazenave. SHPE: HTN planning for video games. In *Wksp. on Computer Games*, pages 119–132. Springer, 2014.

[Musliner *et al.* 2008] David Musliner, Michael J.S. Pelican, Robert .P. Goldman, Kurt D. Kresbach, and Edmund H. Durfee. The evolution of CIRCA, a theory-based AI architecture with real-time performance guarantees. In *AAAI Spring Symp.: Emotion, Personality, and Social Behavior*, 2008.

[Nau *et al.* 1999] Dana Nau, Yue Cao, Amnon Lotem, and Hector Munoz-Avila. SHOP: Simple hierarchical ordered planner. In *Proc. 16th IJCAI*, pages 968–973, 1999.

[Nau *et al.* 2003] D. Nau, T. Au, O. Ilghami, U. Kuter, J. Murdock, D. Wu, and F. Yaman. SHOP2: An HTN planning system. *JAIR*, 20:379–404, 2003.

[Nau *et al.* 2021] Dana Nau, Sunandita Patra, Mak Roberts, Yash Bansod, and Ruoxi Li. GTPyhop: A hierarchical goal+task planner implemented in Python. In *ICAPS Wksp. on Hierarchical Planning (HPlan)*, 2021.

[Nau 2013a] Dana Nau. Game applications of HTN planning with state variables. In *ICAPS Wksp. on Planning in Games*, 2013.

[Nau 2013b] Dana Nau. Pyhop, version 1.2.2: A simple HTN planning system written in Python. Bitbucket, 2013.

[Neufeld *et al.* 2017] Xenija Neufeld, Sanaz Mostaghim, Dario L Sancho-Pradel, and Sandy Brand. Building a planner: A survey of planning systems used in commercial video games. *IEEE Transactions on Games*, 11(2):91–108, 2017.

[Patra *et al.* 2019] Sunandita Patra, Malik Ghallab, Dana Nau, and Paolo Traverso. Acting and planning using operational models. In *AAAI*, pages 7691–7698, 2019.

[Patra *et al.* 2020] Sunandita Patra, James Mason, Amit Kumar, Malik Ghallab, Paolo Traverso, and Dana Nau. Integrating acting, planning, and learning in hierarchical operational models. In *ICAPS*, pages 478–487, 2020.

[Patra *et al.* 2021] Sunandita Patra, James Mason, Malik Ghallab, Dana Nau, and Paolo Traverso. Deliberative acting, planning and learning with hierarchical operational models. *Artificial Intelligence*, 299:103523, 2021.

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

[Sacerdoti 1975] E. Sacerdoti. The nonlinear nature of plans. In *IJCAI*, pages 206–214, 1975.

[Sardina *et al.* 2006] Sebastian Sardina, Lavindra De Silva, and Lin Padgham. Hierarchical planning in BDI agent programming languages: A formal approach. In *Proc. 5th AAMAS*, pages 1001–1008, 2006.

[Shivashankar *et al.* 2012] Vikas Shivashankar, Ugur Kuter, Dana Nau, and Ron Alford. A hierarchical goal-based formalism and algorithm for single-agent planning. In *AAMAS*, pages 981–988, 2012.

[Shivashankar *et al.* 2013] Vikas Shivashankar, Ron Alford, Ugur Kuter, and Dana Nau. The GoDeL planning system: A more perfect union of domain-independent and hierarchical planning. pages 2380–2386, 2013.

[Tate *et al.* 1994] Austin Tate, Brian Drabble, and Richard Kirby. O-Plan2: an open architecture for command, planning and control. In Monte Zweben and Mark S. Fox, editors, *Intelligent Scheduling*. Morgan Kaufmann, 1994.

[Tate 1977] Austin Tate. Generating project networks. In *Proc. 5th IJCAI*, pages 888–893, 1977.

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

[Yao *et al.* 2021] Yuan Yao, Natasha Alechina, Brian Logan, and John Thangarajah. Intention progression using quantitative summary information. In *Proc. 20th AAMAS*, pages 1416–1424, 2021.