GTPyhop: A Hierarchical Goal+Task Planner Implemented in Python

Dana Nau,1,2 Yash Bansod,2 Sunandita Patra,1 Mark Roberts,3 Ruoxi Li1

1Dept. of Computer Science and 2Institute for Systems Research, U. of Maryland, College Park, MD, USA
3The U.S. Naval Research Laboratory, Code 5514, Washington, DC, USA
1,2{nau, yashb, patras, rli12314}@umd.edu, 3{first.last}@nrl.navy.mil

Abstract
The Pyhop planner, released in 2013, was a simple SHOP-style planner written in Python. It was designed to be easily usable as an embedded system in conventional applications such as game programs. Although little effort was made to publicize Pyhop, its simplicity, ease of use, and understandability led to its use in a number of projects beyond its original intent, and to publications by others.

GTPyhop (Goal-and-Task Pyhop) is an extended version of Pyhop that can plan for both goals and tasks, using a combination of SHOP-style task decomposition and GDP-style goal decomposition. It provides a totally-ordered version of Goal-Task-Network (GTN) planning without sharing and task insertion. GTPyhop’s ability to represent and reason about both goals and tasks provides a high degree of flexibility for representing objectives in whichever form seems more natural to the domain designer.

1 Introduction
Pyhop1 is a simple HTN planner written in Python, comprising less than 150 lines of Python code. Its planning algorithm is based on the one in SHOP (Nau et al. 1999), but it avoids the need for a specific “planning” language by having the task network and its methods written directly in Python. Pyhop’s development was motivated by an observation that application developers were often writing planning systems themselves, rather than learning specialized AI planning languages (Nau 2013). Pyhop was written in hopes of providing an HTN (Hierarchical Task Network) planner that could be easily understood by non-AI practitioners.

Pyhop’s author made little effort to publicize it, but the ease with which it could be understood and used has made it useful for rapid prototyping, leading to its use in a number of projects and publications by others (see Section 2).

This paper describes GTPyhop, which extends Pyhop to plan for goals as well as tasks. It combines aspects of both HTN planning as in Pyhop and SHOP, and HGN planning as in GDP (Goal Decomposition Planner) (Shivashankar et al. 2012). In the terminology of (Alford et al. 2016b), it does a totally-ordered version of GTN planning without sharing and task insertion (there is an example at the end of Section 3).

GTPyhop’s source code is about four times as big as Pyhop’s. It includes the following features:

- Rather than a task list, GTPyhop has a to-do list that contains zero or more actions, tasks, and goals. Like Pyhop, it decomposes tasks using task methods; and like GDP, it decomposes goals using goal methods. However, all methods return to-do lists, rather than Pyhop’s task lists or GDP’s goal lists (see example at end of Section 3). Thus a planning domain may use any arbitrary combination of task decomposition and goal decomposition.

- Since HGN planning semantics corresponds readily to classical goal semantics (Shivashankar et al. 2012), it can be used to guarantee soundness. To enforce soundness, when GTPyhop decomposes a goal g, it verifies whether the resulting plan actually accomplishes g, and backtracks if g is not accomplished. We anticipate that in future work, this may be useful for purposes such as verification and validation of domain descriptions.

- GTPyhop can load multiple planning domains into memory, and switch among them without having to restart Python each time. GTPyhop also includes more documentation than Pyhop, and additional debugging features.

The GTPyhop software distribution is available for download under an open-source license.2 In addition to GTPyhop, it includes several example domains, a test harness for running them, and a simple example of planning-and-acting integration: a version of the Run-Lazy-Lookahead actor (Ghallab, Nau, and Traverso 2016) that uses GTPyhop as its planner.

The paper is organized as follows. We provide some context for the original version of Pyhop (Section 2), describe GTPyhop (Section 3), and briefly describe several research projects in which GTPyhop is being used and extended (Section 4). This is followed by discussions of related work (Section 5), some of GTPyhop’s limitations (Section 6), and concluding remarks (Section 7).

2 Why does Py Hop?
As we mentioned earlier, Pyhop is basically a simplified version of SHOP that uses Python syntax. For example, actions and methods are written directly as Python functions. Their preconditions are Python if tests, and their effects are their returned values.

1https://bitbucket.org/dananau/pyhop/src/master/
2https://github.com/dananau/GTPyhop
According to (Nau 2013), the original motivation for Pyhop was a workshop on AI in games (Lucas et al. 2012) in which many of the attendees were developing games that incorporated AI planners as subsystems. The approach was like the way AI planning has been used in other systems that operate in dynamically changing environments:

- Approximate a part of the system’s objective as a planning problem \( p \), and develop a special-purpose planner for \( p \).
- Use the planner as a subroutine, calling it repeatedly to replan as the world changes. The planner operates online, tightly integrated with the rest of the system.

Such integration is easier if the planner is small, easily understandable, and uses data structures compatible with those used in the larger system in which the planner is embedded, rather than requiring the data to be translated between two different representation schemes. Pyhop was written as an example of how to facilitate such integration.

The only efforts to publicize Pyhop were a brief announcement on the SHOP web site\(^3\) and an invited workshop talk (Nau 2013) with no published paper, just slides. Despite this, a Google Scholar search\(^4\) shows 66 publications that refer to Pyhop. In many of them, Pyhop is used for applications having nothing to do with games. There also have been several forks of Pyhop, e.g., (McGreggor 2014; Cheng et al. 2018), and a re-implementation of Pyhop in C++ (Jaccopin 2020).

### 3 GTPyhop

Figure 1 shows the GTPyhop planning algorithm. Like Pyhop, it does a depth-first search with no loop detection (which wouldn’t be useful here, see Alford et al. (2012)). Compared to Pyhop, there are the following extensions:

1. **refine-goal-and-continue** does HGN decomposition based on the GDP algorithm (Shivashankar et al. 2012).
2. A mixture of task decomposition and goal decomposition may be used throughout a GTPyhop planning domain. In the to-do lists \( T \) and \( T_{\text{sub}} \) in lines (ii), (ii), and (iii), each element may be a task, a goal, or an action.
3. In line (iii), \( g \) is a goal, so GTPyhop needs to ensure that \( T_{\text{sub}} \) achieves \( g \). To do so, it appends to \( T_{\text{sub}} \) a dummy action \( \text{verify}(g) \) that has \( g \) as a precondition. If the state produced by \( T_{\text{sub}} \) satisfies \( g \), the action has no effect. Otherwise the action fails, making GTPyhop backtrack.

#### 3.1 Representations and examples

We now describe the basic elements of a GTPyhop planning domain, with examples from the GTN blocks-world domain included with the GTPyhop software distribution.\(^5\)

**Domains** are Python objects that contain all the elements of a planning domain, e.g., `gtpyhop.Domain('blocks_gtn')`.

**States** are Python objects that contain collections of state-variable bindings. When one first defines a state \( s \), it is easiest to write the variable bindings in dictionary form, as in

```python
GTPyhop(s0, T)
return seek-plan(s0, T, [])  # base case for seek-plan
```

```
seek-plan(s, T, π)
# recursive DFS; π is the current partial solution
if T = [] then return π

T' ← the rest of T

 case(t): # solve t, then plan for T'
   action: return apply-action-and-continue(s, t, T', π)
   task: return refine-task-and-continue(s, t, T', π)
   goal: return refine-goal-and-continue(s, t, T', π)

apply-action-and-continue(s, a, T', π)
if action \( a \) is applicable in state \( s \):
  if no solution exists.
  Actions and methods are functions, so \( a(s) \) is the state produced by \( a \), and \( m(s, t) \) is the to-do list returned by \( m \).

"+" denotes concatenation of lists.

Figure 1: GTPyhop pseudocode. The arguments are the initial state \( s_0 \) and a to-do list \( T \) (a list of actions, tasks, and goals). GTPyhop returns a solution plan \( π \), or failure if no solution exists. Actions and methods are functions, so \( a(s) \) is the state produced by \( a \), and \( m(s, t) \) is the to-do list returned by \( m \).

**Actions** and **methods** are written as ordinary Python functions, with the current state as the first argument. There isn’t a special reasoning system (e.g., SHOP’s Horn-clause inference) to evaluate an action’s or method’s preconditions. Instead, preconditions are evaluated using ordinary Python if tests. Similarly, an action’s effects, and the to-do list returned by a method, are produced by ordinary Python computations.

For example, Figure 3 shows the blocks-world pickup action. Its arguments are the current state \( s \) and the block \( x \) to pick up. The precondition is the first if test: if \( x \) is clear, \( x \) is on the table, and the robot hand is empty, then the action modifies the state to say that \( x \) is in the robot hand, and returns the modified state. Otherwise the action returns no value, which tells GTPyhop the action is inapplicable.

**Tasks and task methods**: Tasks are represented as tuples. For example, let (‘take’,\( x \)) be the task of picking up a block

---

\(^3\)https://www.cs.umd.edu/projects/shop/
\(^4\)https://scholar.google.com/scholar?hl=en&q=pyhop
\(^5\)https://github.com/dananau/GTPyhop/tree/main/Examples/blocks_gtn
def m_moveblocks(s, mgoal):
    for x in all_clear_blocks(s):  # find a block to move
        stat = status(x, s, mgoal)
        if stat == 'move-to-block':
            where = mgoal.pos[x]  # where to move it
            return [('take', x), ('put', x, where), mgoal]  # (iv)
        elif stat == 'move-to-table':
            return [('take', x), (put, x, 'table'), mgoal]  # (v)
        for x in all_clear_blocks(s):  # resolve deadlock
            if status(x, s, mgoal) == 'waiting' \
                and s.pos[x] == 'table':
                return [('take', x), ('put', x, 'table'), mgoal]  # (vi)
    return []  # no blocks need to be moved

gtpyhop.declare_multigoal_methods(m_moveblocks)

Figure 5: A multigoal method implementing the block-stacking algorithm in (Gupta and Nau 1992): if a block needs
moving and is ready to move to its final location, then do so and
continue planning for mgoal; else if there’s a deadlock then resolve
it and continue planning for mgoal; else we’re done.
The code uses two helper functions: all_clear_blocks
returns a list of clear blocks, and status tells whether a block
x needs to be moved and whether it is ready to be moved.

A GTN planning example. In lines (iv), (v), and (vi) of
Figure 5, each to-do list contains two tasks and a multigoal.
GTPyhop knows what kind of method to use in each case.

4 Example Usages

There are several research projects in which GTPyhop is
being used and extended. We briefly describe them below.

Bansod et al. (2021) describes an integrated system for
hierarchical planning and acting in dynamically changing
environments. An important component of this system is a
re-entrant planning algorithm based on GTPyhop.

A paper in preparation integrates GTPyhop with reinforce-
ment learning. The work uses the goal network provided by
GTPyhop to guide a curricula for multi-task learning. During
acting, it executes the goal network provided by GTPyhop.

GTPyhop is also being used in a research project to develop
temporal planning algorithms in multi-agent environments.
For this purpose, modifications are being made to support
communication among multiple agents, and representation
and reasoning about temporal constraints.

In our future work, we anticipate the possibility of using
goals for verification and validation of domain descriptions.

5 Related Work

The closest related theoretical work is (Alford et al. 2016b),
which related task networks and goal networks under various
semantics, including HTN, HGN, and GTN planning, task
(or goal) insertion, and sharing.

Several HTN planners introduced in the 1970s through
1990s are no longer available for comparison. One of the first
was NOAH (Sacerdoti 1975), which was followed by Nonlin
Many HTN planners provide a planner-specific language in which to write the HTN methods. The SHOP planners (Nau et al. 1999, 2003; Goldman and Kuter 2019) make use of Lisp’s extensibility to define a Lisp-like language for this purpose. Sohrabi et al. (2009) extended PDDL3 with HTNs to support preferences and converted this extended PDDL3 format for a variant of SHOP2.

In JSHOP2 (Ilghami and Nau 2003), methods are written in the same Lisp-like language, but JSHOP2 compiles them to Java to perform the search. Similarly, the HyperTensioN planner converts a planning model into the Ruby language and was recently extended to support semantic attachments for HTN (Magnaguagno and Meneguzzi 2020).

The totSAT planner (Behnke, Höller, and Biundo 2018) converts totally-ordered HTN planning problems into a SAT formula. PANDA (Höller et al. 2021) is a planner that integrates various approaches to hierarchical planning. Both planners emphasize the importance of a common language for problem definition and propose a Hierarchical Domain Definition Language (HDDL) for it (Höller et al. 2020).

HDDL was also the input language for the recent Hierarchical Track of the International Planning Competition. A full discussion of all planners in that competition is out of scope for a short paper, but we highlight the winners SIADEX (de la Asunção et al. 2005; Castillo et al. 2005) and HyperTensioN (Magnaguagno and Meneguzzi 2020), which both translated HDDL to their specific format.

The IxTeT planner (Ghallab and Laruelle 1994) was a temporal HTN planner which used a specialized language to encode its methods. OpenPRS is a C implementation of PRS. ASPEN (Fukunaga et al. 1997; Chien et al. 2000) is both a planner and framework for planning in space applications; it uses a planner-specific language for encoding plans. The FAPE Planner (Dvorák et al. 2014; Bit-Monnot et al. 2020) is a recent planner that supports a subset of the ANML language (Smith, Frank, and Cushing 2008). A more recent HTN planner SHPE (Menif, Jacopin, and Cazenave 2014) is specifically developed for AI planning in video games. It used a simplified variant of ANML (Smith, Frank, and Cushing 2008) to encode problems that were then compiled C++ objects to perform search. The Adversarial HTN Planner (Ortaño and Buro 2015) allows for HTNs to be used in iterated environments such as Real Time Strategy games; problems for this planner are encoded in a language provided by the system and a variety of research has extended this planner (e.g., (Lin et al. 2020; Sun et al. 2017)).

Like Pyhop and GTPyhop, there are several HTN planners in which domains and problems are written in a conventional programming language. The planner by Neufeld et al. (2018) uses C++ for this purpose; its HTN primitives link with Behavior Trees, a common representation for computer game agents. The planner by Soemers and Winands (2016) also uses C++ to represent HTN problems; this planner introduced a mechanism to reuse the existing solution for faster replanning. The UPOM planner introduced in (Patra et al. 2020) uses Python to represent hierarchical operational models.

6 Limitations

One limitation involves the goal representation’s expressivity. A GTPyhop goal, like a state (see Figure 2), is a set of state-variable bindings. The goal is the conjunction of those bindings, without a way to represent more complicated logical expressions. There probably are some workarounds, but we have not yet considered this. In our work so far, this limitation has not been a major problem.

In many HTN planners, a method or action may contain free variables for which there are several possible instantiations. When the planner creates instances of the method or action, it may backtrack over these instantiations. In contrast, in GTPyhop (like Pyhop) there is no notion of instantiating a method. The method is a piece of Python code that GTPyhop calls directly. The method may contain a variety of local variables, but it is up to the method’s author to specify how these variables will acquire their values.

In HTN planners that use planner-specific languages, the methods’ preconditions and subtasks are data structures that the planner can reason about before deciding which methods to use in a planning problem. This has enabled several recent advances in HTN-planning search heuristics (Alford et al. 2016a) and other speedup techniques (Behnke, Höller, and Biundo 2018). In contrast, GTPyhop does not know its methods’ preconditions and subtasks in advance, because each method is a Python program that computes a list of subtasks and subgoals that may depend on the current state (e.g., Figures 4 and 5). A potential way to circumvent this limitation might be to evaluate the method, see what tasks, goals, and actions it returns, and then use this information to provide input to a search heuristic—but we have not tried to implement this to see how well it would work. For now, GTPyhop (like Pyhop and SHOP) just does a depth-first search, trying methods in the order that the domain author defined them.

7 Conclusions

Pyhop implemented a version of SHOP-style HTN planning in which methods and actions were written directly in Python. Despite a minimal amount of publicity and no publication, it has been used in several systems that went beyond its original intent of a simple planner for game systems. GTPyhop extends Pyhop to provide a version of totally-ordered Goal-Task-Network planning without sharing and task insertion. GTPyhop also includes several other features, as described in the introduction. We are working now to extend GTPyhop to incorporate temporal and multi-agent concerns. Section 4 has briefly described the directions that this work is taking.

Acknowledgments. Many thanks to the anonymous reviewers for their insightful remarks. This work has been supported in part by NRL grants N00173191G001 and N0017320P0399, and ONR grant N000142012257. The information in this paper does not necessarily reflect the position or policy of the funders, and no official endorsement should be inferred.
References


Behnke, G.; Höller, D.; and Biundo, S. 2018. totSAT – totally-ordered hierarchical planning through SAT. In AAAI.


