Efficiently Storing States For HTN Planner Backtracking

Tyler Kitts[1]

[1] Dept. of Computer Science, Univ. of Maryland, College Park, MD, USA

Abstract

Abstract When a HTN planner makes a choice which leads to a dead end, it may backtrack to a previous state to choose a different option. This requires the planner to copy and store every state it creates for later backtracking, however copying the current state is costly. To mitigate this issue we modified IPyHOP to avoid storing information that is the same in multiple stored states. This saves time that would have been spent copying the state by instead creating pointers to the variables which are the same in both the new state and a older state.

1. Introduction

Pyhop and its successor GTPyhop are HTN planners which use a description of the world in the form of possible actions, a set of variables describing the current state of the world, and a set of variables describing the goal state. GTPyhop uses that goal state as the basis for a to-do list which is broken down into sequences of tasks or actions [3]. When GTPyhop generates a plan that does not accomplish the goal it backtracks to the last state checks if it can produce a plan that accomplishes the goal. If it cannot then GTPyhop repeats this backtracking until it finds a state that leads to accomplishing the goal.

IPyHOP is a re-entrant iterative HTN planner written in Python and based on GTPyhop. IPyHOP is based on GTPyhop, but can resume planning at the point where the failure occurred [1]. To facilitate this backtracking IPyHOP has to store every state it generates during planning in case it has to backtrack to that state later.

The Simple Hierarchical Ordered Planner (SHOP) planning algorithm is a HTN planner written in Lisp which predates Pyhop. SHOP2 and SHOP3 [2]. SHOP3 tracks the changes to the current state through separate incremental updates to an original state; These incremental updates can then be rolled back one at a time to backtrack to previous states. This method of tracking changes to a state provided the inspiration for improving the efficiency of IPyHOP's backtracking. Due to IPyHOP being iterative and SHOP3 being recursive this modified backtracking is different, though more directly analogous backtracking method could be applied to GTPyhop in future work.

Our primary contribution is a way to increase the efficiency of IPyHOP's backtracking by reducing the number of variables that must be copied from a state to prepare for later backtracking. This paper discusses provides further details on GTPyhop, IPyHOP and SHOP3 in Section 2, Section 3 explains the modifications made to IPyHOP to improve backtracking, Section 4 covers two domains for HTN planning and the results of experiments comparing IPyHOP and the modifications we made, and Section 5 discusses the limitations of our work and features that future work could expand on.

Algorithm 2: IPyHOP's HTN pseudocode.

```
1  IPyHOP(current state s, decomposition tree w):
2    p ← w's root
3    loop
4      if all tasks in w have been expanded then return w
5      u ← the first un-expanded node of w
6      if u has been visited then
7        s ← state(u)          # use the cached state
8      else state(u) ← s       # cache the current state
9      mark u as visited
10     t ← task(u)
11     if t matches an operator o then
12       if o is applicable in s then
13         s ← the state produced by applying o to s
14         mark u as expanded
15     else
16       for each method m that matches t
17         if m hasn't been already tried for t
18           if m is applicable in s then
19             mark u as expanded
20             install m's subtasks as children of u
21             exit the for loop
22     if u hasn't been expanded then
23       backtrack(w, u)
24 backtrack(w, u):
25   v ← non-primitive task node expanded before u
26   un-expand all nodes expanded after and including v
```

Figure 1. Pseudocode for IPyHOP [1]

2. Background: GTPyhop, IPyHOP and SHOP3

HTN planners are domain configurable planners that refine a set of tasks to produce a plan. These planners require a description of the world in the form of a set of actions for interacting with the world and a set of variables which are used to describe a state.

GTPyhop (Nau et al. 2021) is a domain-independent planner written in Python. GTPyhop recursively plans for a set of tasks and goals in the same order that they will later be executed. GTPyhop is the basis for both IPyHOP and out modification of IPyHOP[3].

IPyHOP is a planning system written in Python as an iterative version of GTPyhop. GTPyhop recursively refines goals and tasks, but this can make re-entering due to action failure impossible. IPyHOP was made to improve GTPyhop's ability to replan by being an iterative version which stores states as nodes in a tree. IPyHOP maintains a hierarchy of state nodes so that it can move to a different node when it encounters a failure. Line 8 of IPyHOP's pseudocode, which caches the current state, occurs very often so our improvement to this caching is triggered many times[1].

SHOP3 is a HTN planning system written in Lisp and based on SHOP2. SHOP3 modernized SHOP2's architecture to make incorporating SHOP into external systems easier and support Planning Domain Definition Language (PDDL). SHOP3 reads PDDL files and breaks the effects of each action into individually traceable changes which it can revert during backtracking. This way the entire state of each node does not need to be stored, instead only the exact changes to individual variables within a state are stored and undone for backtracking[2].

3. Efficiently storing states

Our contribution is an improvement to IPyHOP's backtracking procedure. After IPyHOP expands a new node by selecting an action it updates the current tracked state to match the results of that action, and it stores the current state as a node (line 8 of figure 1). To store that state as a node in a tree it performs a deep copy of each dictionary stored at that node. For the Satellite domain a node would include a dictionary for the direction each satellite is facing, which instruments are powered on, which satellites have power available, which instruments are calibrated, and what images have been taken. For any domain where actions may not affect every dictionary, performing a deep copy is slow and uses more memory than necessary. Similarly

Problem Size	Base IPyHOP	Modified IPyHOP
5	2.20E-03	4.14582E-05
10	3.74E-03	3.92356E-05
15	6.87E-03	5.48842E-05
20	1.49E-02	6.92645E-05
25	1.80E-02	6.44468E-05
30	2.15E-02	6.90166E-05
35	2.81E-02	9.97501E-05
40	3.39E-02	1.02E-04

Figure 2. Mean time to copy a state in the rover domain. A problem size of X means there are X rovers, X objectives for imaging, X cameras, X goals, and X+5 waypoints

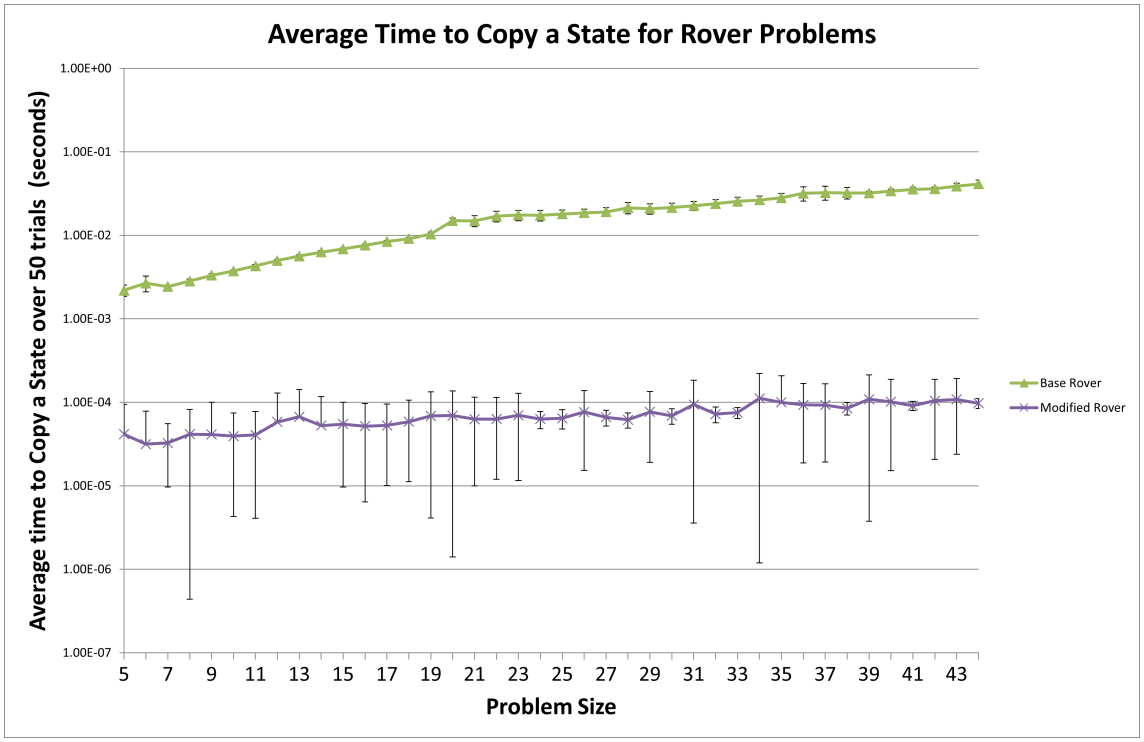

Figure 3. Time required to copy one state by IPyHOP in the Rover domain on a semi-log scale. The error bars show the standard error. One state contains the following dictionaries: empty rovers, rovers carrying rock analysis, rovers carrying soil analysis, full rovers, calibrated rovers, rovers that have an image, communicated soil data, communicated rock data, communicated image data, location of soil samples, location of rock samples, free communication channels, objectives at each waypoint.

to how IPyHOP requires a list of declared actions, our modified version of IPYHOP requires a dictionary that maps actions to the variables they can affect. Our modified version of IPyHOP uses this dictionary to identify which variables were modified by an action, and thus need to be deep copied, and which variables can be replaced with a pointer referencing the parent node's corresponding variable. For example, when an instrument is calibrated in the satellite domain the dictionary that maps each instrument with their calibration status must be deep copied, but that is the only dictionary that is affected by that action so all other variables in the new node are replaced with pointers. The figure below shows a more complex example, where multiple actions are taken.

Rigid Relations

Satellites={Sat1, Sat2}
Directions={Star1, Star2}
Instruments={Instrument1, Instrument2, Instrument3}
On_Board={Instrument1:Sat1, Instrument2:Sat1, Instrument3:Sat2}
Supports={Instrument1:Spectrograph1, Instrument2: Spectrograph2, Instrument3, Spectrograph1}
Calibration_Target={Instrument1:Star1, Instrument2:Star2, Instrument3:Star1

Current Action	Points_To	Power_On	Power_Available	Calibrated	Image
	{Sat1:Star1, Sat2:Star2}	{Instrument1:False, Instrument2:True, Instrument3:False}	{Sat1:False, Sat2:True}	{Instrument1:True, Instrument2:False, Instrument3:False}	{}
Turn_To(satellite2,Star1,star2)	{Sat1:Star1, Sat2:Star1}	↑	↑	↑	↑
Switch_On(Instrument3, Sat2)	↑	{Instrument1:False, Instrument2:True, Instrument3:True}	{Sat1:False, Sat2:False}	↑	↑
Calibrate(Instrument3, Sat2)	↑	↑	↑	{Instrument1:True, Instrument2:False, Instrument3:True}	↑
Take_Image(Instrument3, Sat2, Star2, Spectrograph1}	↑	↑	↑	↑	{star2:[Spectrograph1]}

Figure 4. This chart shows what is stored in each dictionaries at a node when it is generated by taking an action. The leftmost column shows the most recent action; the first row shows the name of each dictionary; the second row shows the initial values for each dictionary; each subsequent row shows what is stored at the newly generated node. Arrows signify a pointer to the previous node's dictionary for that variable. At the top of the figure is a list of rigid relations, which are dictionaries that do not need to be copied because they cannot be modified.

4. Experiments

We expect our modified IPyHOP to be more efficient than base IPyHOP because it copies fewer variables when cloning a state node. We expect a greater difference in more complex domains where each action only affects a small subset of the variables that make up a state.

4.1. Satellite Domain

This Satellite domain was derived from the IPC 2002 competition where images of stars and planets need to be taken using different instruments housed on separate satellites. For this evaluation 50 trials were performed for 50 problem sizes. A problem size of X means there are X satellites, X maximum instruments per satellite, X imaging modes, X imaging targets and X required observations. Planning in the Satellite Domain with modified IPyHOP requires significantly less time and memory than base IPyhop, and both planners produce similarly sized plans. These results suggest modified IPyHOP is an improved algorithm since it is more efficient at producing the same plans.

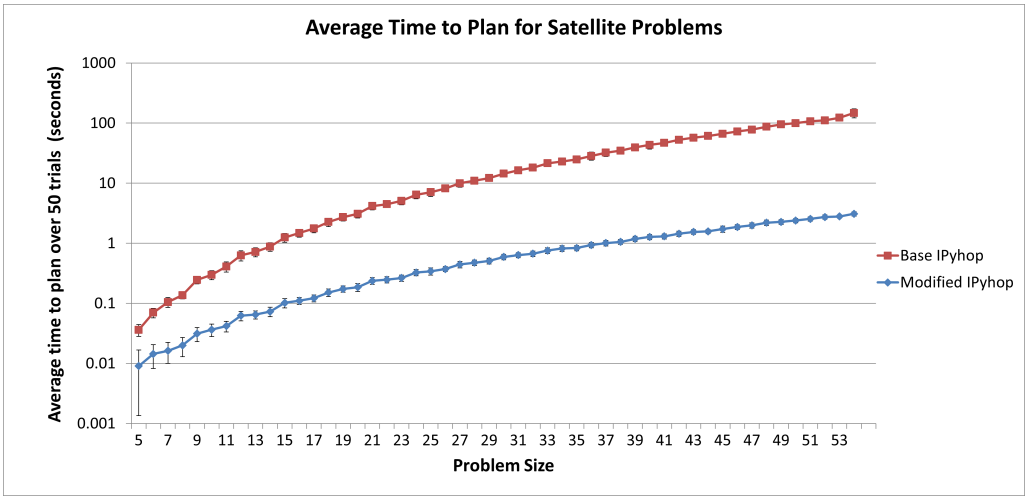

Figure 5. Mean time spent to plan by modified and unmodified IPyHOP in the Satellite domain on a semi-log scale. The error bars show the standard error.

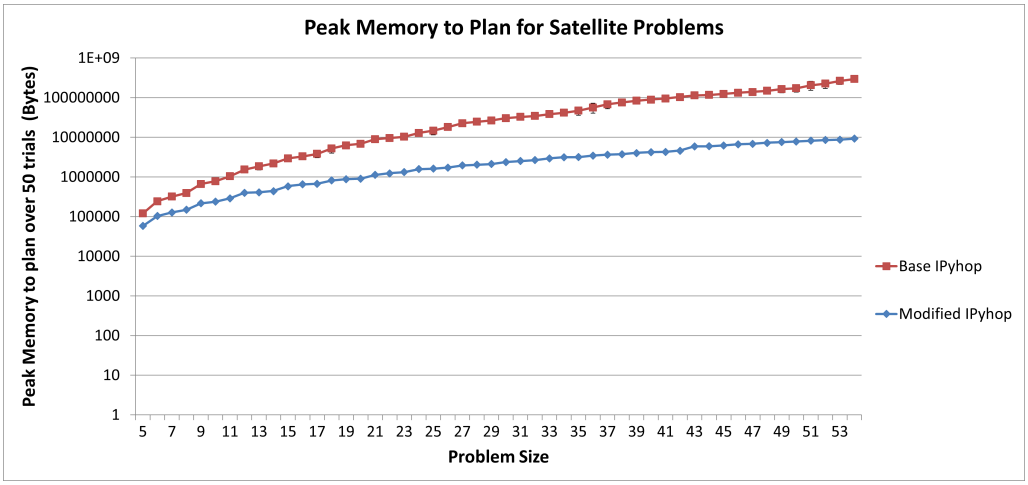

Figure 6. Peak Memory usage of modified and unmodified IPyHOP in the Satellite domain on a semi-log scale. The error bars show the standard error.

4.2. Rover Domain

This Rover domain was derived from IPC 2002 competition where rovers must collect samples from waypoints and maneuver between locations to take images and communicate its collected data. For this evaluation 50 trials were performed for 40 problem sizes. A problem size of X means there are X rovers, X objectives for imaging, X cameras, X goals, and X+5 waypoints. Similarly to the Satellite domain modified IPyHOP outperformed base IPyhop in both mean time required to plan and peak memory usage while producing similar plans.

5. Results and Discussion

This improvement to IPyHOP was inspired by SHOP3's approach to storing states which makes undoing actions less computationally expensive. SHOP3 stores the changes an action causes as a tag which can be applied or removed from a state, but an iterative program like IPyHOP which maintains a hierarchy of nodes can point to other nodes rather than store duplicate data. This modification performed well in the Satellite Domain and Rover domain, especially for larger problems. While the time required to plan still follows polynomial growth as problem size increases for both Domains, but it is a slower polynomial growth.

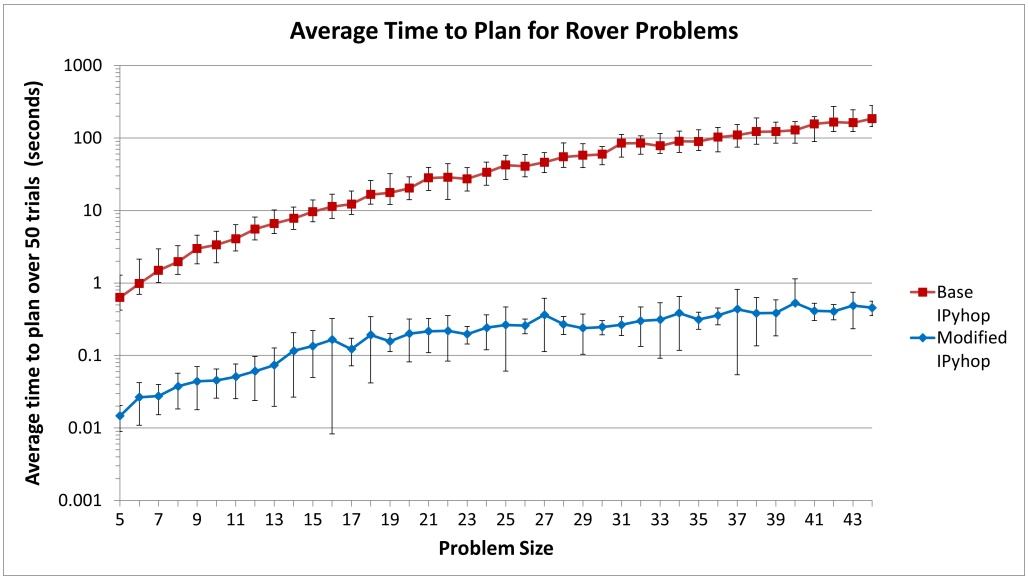

Figure 7. Mean time spent to plan by modified and unmodified IPyHOP in the Rover domain on a semi-log scale. The error bars show the standard error.

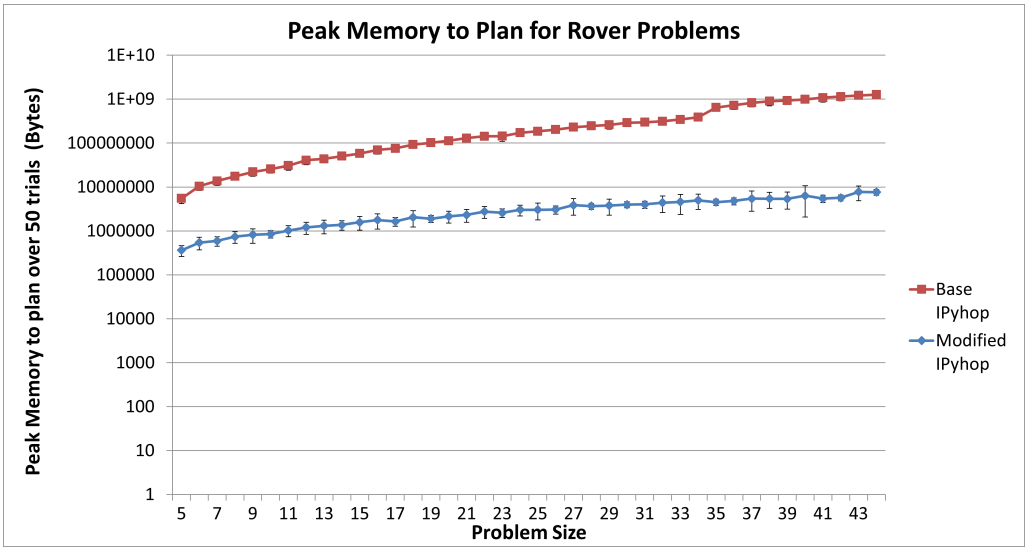

Figure 8. Peak Memory usage of modified and unmodified IPyHOP in the Rover domain on a semi-log scale. The error bars show the standard error.

Future work could further isolate changes to the state by tracking which items are added or removed to the state. Instead of tracking which variables are modified and only copying those, SHOP3 tracks the modifications and when its planner has to backtrack it recreates the state by applying a list of tracked changes to an original state. Doing so in an iterative planner that stores a hierarchy of nodes complicates backtracking to an arbitrary node, but it would further improve efficiency. A future work which recreates SHOP3's change tracking system could disassemble the effects of actions listed in a PDDL file in a way that identifies which keys match modified data in dictionary variables or which indices identify the modified elements of a list.

Acknowledgements

This paper would not have been possible without the guidance of Professor Dana Nau. We would also like to thank Mark Roberts for assistance proof reading and providing feedback on this

paper and Paul Zaidins for providing his insight into IPyhop and the Rover domain.

References

[1] Yash Bansod, Sunandita Patra, Dana Nau, and Mark Roberts. HTN replanning from the middle. In *FLAIRS*, May 2022.

[2] Robert P Goldman and Ugur Kuter. Hierarchical task network planning in Common Lisp: the case of SHOP3. In *Proc. European Lisp Symposium*, 2019.

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