ABSTRACT

Title of Thesis: REFINEMENT ACTING VS. SIMPLE EXECUTION GUIDED BY HIERARCHICAL PLANNING

Yash Bansod
Master of Science, 2021

Thesis Directed by: Professor Dana Nau
Department of Computer Science
and Institute of Systems Research

Humans have always reasoned about complex problems by organizing them into hierarchical structures. One approach to artificial intelligence planning is to design intelligent agents capable of breaking complex problems into multiple levels of abstraction so that at any one level, the problem becomes small and simple. However, for an agent to reason at multiple levels of abstraction, it needs knowledge at those abstraction levels. Hierarchical Task Network (HTN) planning allows us to do precisely that. This thesis presents a novel HTN planning algorithm that uses iterative tree traversal to refine HTNs. We also develop a purely reactive HTN acting algorithm using a similar procedure. Preserving the hierarchy in HTN plans can be helpful during execution. We make use of this fact to develop an algorithm for integrated HTN planning and acting. We show through experiments that our algorithm is an improvement over a widely used approach to planning and control.

Refinement Acting vs. Simple Execution Guided by Hierarchical Planning

by

Yash Bansod

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2021

Acknowledgments

I would like to thank my thesis advisor, Professor Dana Nau, for his guidance and mentorship throughout my thesis research. He established an environment where I was encouraged to freely express my ideas and discuss them without hesitation. I am especially thankful for everything he has taught me and the wonderful technical discussions we had, which were crucial to this thesis research. He has always been patient with me, welcomed my opinions, and inspired me to devise creative ways of solving problems.

I am also thankful to Dr. Sunandita Patra for her assistance. This research would not have been possible without her valuable feedback on my work. The technical discussions that we had helped cement many concepts used for the development of this thesis.

I am incredibly grateful to Dr. Muralikrishna Sridhar, who encouraged me to pursue thesis research for my master's education. I value his mentorship a lot, and I thank him for his role in sculpting me as my present self.

Lastly, I am thankful to my parents for always encouraging me and supporting me in my pursuit to improve myself. And for always expecting higher standards from me than I ever did for myself. This work would have been impossible without the sacrifices they made to shield me from all the hindrances I would have faced otherwise.

Table of Contents

Acknowledgements ii

Table of Contents iii

List of Tables v

List of Figures vi

List of Abbreviations vii

Chapter 1: Introduction 1
- 1.1 Motivation . . . 3
- 1.2 Contributions of the Thesis . . . 5
- 1.3 Thesis Organization . . . 7

Chapter 2: Related Work 9
- 2.1 AI Planning . . . 9
- 2.2 AI Acting . . . 10
- 2.3 Integrating AI planning and acting . . . 11

Chapter 3: Planning and Acting Algorithms 13
- 3.1 HTN Planning Formulation . . . 14
- 3.2 HTN Planning in Pyhop . . . 18
- 3.3 HTN Planning in IPyHOP . . . 20
- 3.4 RAE-lite - A purely reactive HTN Actor . . . 23
- 3.5 Integrating IPyHOP with an Actor . . . 26
  - 3.5.1 Run-Lazy-Lookahead . . . 26
  - 3.5.2 Run-Lazy-Refineahead . . . 30

Chapter 4: Experimental Evaluation 36
- 4.1 RoboSub Domain . . . 36
- 4.2 Experimental Setup . . . 40
- 4.3 Results . . . 42
- 4.4 Summary . . . 50

Chapter 5: Conclusion 53
- 5.1 Limitations and Future Work . . . 55

Appendix A: RoboSub Domain in IPyHOP 58
A.1 Method Definitions 58
A.2 Action Definitions 63

Bibliography 68

List of Tables

4.1 Overview of results obtained using D_{mean_exp} 50
4.2 Overview of results obtained using $D_{equivalent}$ 50

List of Figures

1.1 Deliberative acting architectures: (a) Planner uses descriptive models while actor uses operational models; (b) Planner and actor both use the same operational models. 4

3.1 Task network visualizations: (a) Visualization of a task, method, preconditions, sub-tasks, and operators. (b) Compressed visualization of a task network used for visualizing larger networks. 16
3.2 Actor architectures: (a) Interaction of RAE with execution platform. (b) Interaction of RAE-lite with execution platform. 24
3.3 Refinement of task $t1$ using $m1\_t1$ and $m2\_t1$. And refinement of task $t2$ using $m1\_t2$ and $m2\_t2$. (Example 1). 28
3.4 Task network visualizations: (a) After first planning attempt. (b) Re-planning after failure in execution of o6. 28
3.5 (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). 32

4.1 An illustration of a sample planning problem for the RoboSub 2019 competition. 37
4.2 Results of metric - Total nodes expanded (a) Distribution visualized using histograms (b) Relation visualized by fitting a line on the scatter plot. 45
4.3 Results of metric - Total actions planned (a) Distribution visualized using histograms (b) Relation visualized by fitting a line on the scatter plot. 46
4.4 Results of metric - Total iterations taken (a) Distribution visualized using histograms (b) Relation visualized by fitting a line on the scatter plot. 47
4.5 Results of metric - Total action cost (a) Distribution visualized using histograms (b) Relation visualized by fitting a line on the scatter plot. 48
4.6 Results of metric - Final state reward (a) Distribution visualized using histograms (b) Relation visualized by fitting a line on the scatter plot. 49

List of Abbreviations

A-H

AI	Artificial Intelligence
AUV	Autonomous Underwater Vehicle
BFS	Breadth First Search
DFS	Depth First Search
GNPyhop	Goal Network Python Hierarchical Ordered Planner
GOAP	Goal Oriented Action Planning
HGN	Hierarchical Goal Network
HTN	Hierarchical Task Network

I-R

IPyHOP	Iterative Python Hierarchical Ordered Planner
NOAH	Nets of Action Hierarchies
O-Plan	Open Planning Architecture
PDDL	Planning Domain Definition Language
PRS	Procedural Reasoning System
Pyhop	Python Hierarchical Ordered Planner
RAE	Refinement Acting Engine
RAE-lite	Refinement Acting Engine lite

S-Z

SeRPE	Sequential Refinement Planning Engine
SHOP	Simple Hierarchical Ordered Planner
SHPE	Simple Hierarchical Planning Engine
SIPE	System for Interactive Planning and Execution
STRIPS	Stanford Research Institute Problem Solver
UMCP	Universal Method Composition Planner

Chapter 1: Introduction

Artificial Intelligence (AI) planning or Automated planning is a rich technical field. *Planning* can be defined as an abstract, explicit deliberation process that chooses and organizes actions by anticipating their expected outcomes. In AI planning, we study this deliberation process computationally [1]. In practical applications, however, planning alone is rarely the ultimate objective. It is usually followed by *acting*. The integration of planning and acting, or *deliberative acting* is a crucial area of study in our view. Deliberation for acting consists of deciding which actions to undertake and how to perform them to achieve an objective. It refers to a reasoning process, both before and during acting [2]. Here we study the computational deliberation capabilities that allow an artificial agent to reason about its actions, choose them, organize them purposefully, and act deliberately to achieve an objective.

Planning can mean different things in different contexts. Examples include path and motion planning, perception planning and information gathering, navigation planning, mission planning, manipulation planning, communication planning, and several other social and economic planning forms.

AI Planning is a process whereby a system attempts to figure out a sequence of actions that will achieve a distant goal set upon it by the user. This sequence of actions is called a *plan*.

Classical planning refers generically to planning for restricted state-transition systems. To do this, we model the problem in some language or *encoding* that tells us all the information that can be true about the world at that time. These bits of information are known as facts or *predicates*. Predicates tell us things we might need to know at a later point in time in our planning. We store all of the predicates we have representing how the world looks like at any point in time within a *state*. We design *operators* capable of modifying the state. Operators when instantiated are called *actions*. Actions are usually broken into three parts: the *objects* - things that are involved in the action, the *preconditions* - predicates that must be true before we apply the action, and the *effects* - which represent how the world changes as a result of completing the action, adding new information to the world state or deleting existing facts that are no longer true.

Hierarchical Task Network (HTN) planning [1, Chapter 11] is a branch of AI planning that represents and handles *hierarchies*. In some aspects, HTN planning is like classical planning in that a set of atoms represents each state of the world, and each action corresponds to a deterministic state transition. However, HTN planners differ from classical planners in what they plan for and how they plan for it. In an HTN planner, the objective is not to achieve a set of goals but to perform some set of *tasks*. The input to the planning system includes a set of *operators* similar to classical planning and a set of *methods*, each of which is a prescription for decomposing some tasks into some set of sub-tasks (smaller tasks). Planning proceeds by decomposing *non-primitive* tasks in the initial task network recursively into smaller and smaller sub-tasks until *primitive* tasks are reached. The primitive tasks can be performed directly using the planning operators at the given initial state.

1.1 Motivation

Hierarchies are one of the most common structures used to understand and conceptualize the world. The intuitive hierarchical representation used by HTN planners allows the often available expert knowledge about a domain to be included with relative ease to guide the search process. This expressive power of HTN methods is beneficial for the development of planners for various practical applications [3–5]. In practice, the inclusion of such search control knowledge can make HTN planning faster than classical planning, and a good set of methods can enable an HTN planner to perform well on benchmark problems [6]. HTN planning has especially seen wide adoption for mission planning in robotics [4, 5] and Game AI development [3, 7].

In AI planning and acting, there are two main ways of defining action models, descriptive and operational models. Descriptive models of actions specify the state or set of possible states that may result from performing an action. In contrast, operational models describe how to perform an action: what commands to execute in the current context and how to organize them to achieve the action's intended effects.

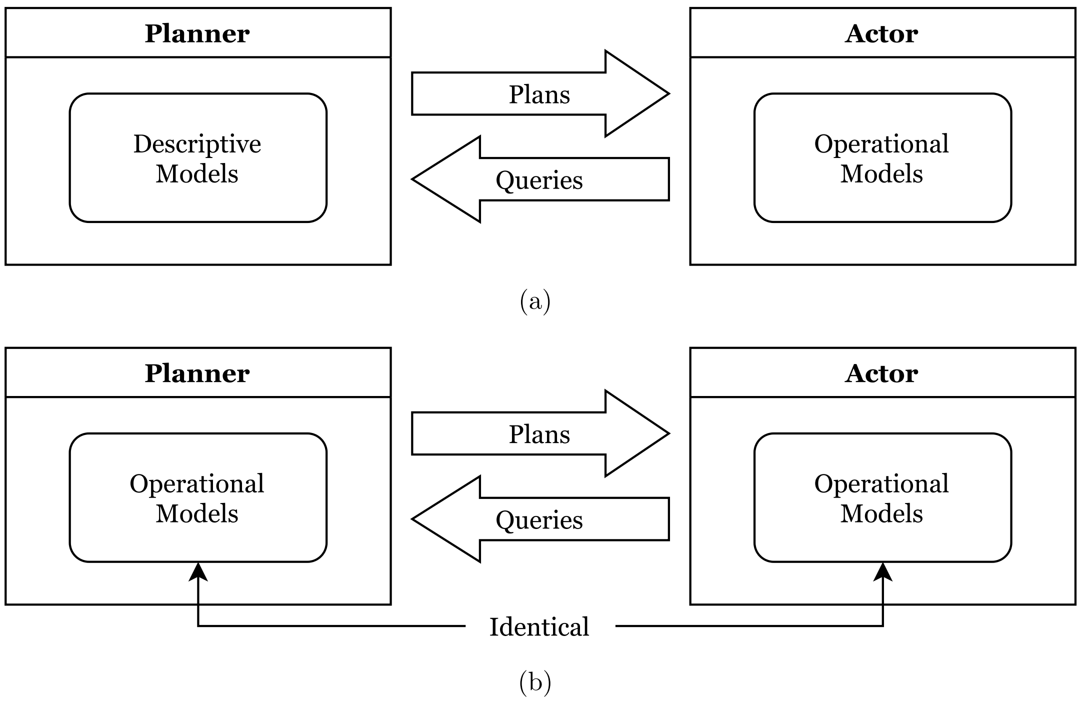


Figure 1.1: Deliberative acting architectures: (a) Planner uses descriptive models while actor uses operational models; (b) Planner and actor both use the same operational models.

HTN planners use *descriptive models* of actions tailored to compute the next states in a state transition system efficiently. Plans generated using descriptive models execute well with closed, static, and deterministic world assumptions. However, executing the plan in open, dynamic, and non-deterministic/probabilistic domains (characteristic of many practical problems) eventually 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 due to (i) failure in execution of actions, (ii) occurrence of unexpected events, (iii) because the planning was done with incorrect/partial information, et cetera.

As argued above and by many prominent authors, plans are needed for acting deliberately, but they are not sufficient for deliberative acting [8]. Many deliberative acting approaches seek to combine the descriptive models used by the planner with the *operational models* used by the actor [9]. In contrast, others seek to directly integrate planning and acting using operational representations [10,11]. A schematic diagram showing these approaches is shown in figure 1.1.

We aim to develop an efficient HTN based deliberative acting algorithm.

1.2 Contributions of the Thesis

The most significant contribution towards the popularization of HTN planning has emerged after the proposal of the Simple Hierarchical Ordered Planner (SHOP) [12], and its successors SHOP2 and SHOP3 [13, 14]. However, SHOP and its successors are written in the LISP programming language, which limits its adoption. Python is a much more popular language and is widely adopted by roboticists, game developers, machine learning engineers, and AI engineers. A quick search of the number of repositories using Python vs. the number of repositories using LISP on GitHub makes the popularity of Python apparent. However, there are very few implementations of HTN planners in Python.

One of the most popular implementations is Pyhop[1] [15], a Python adaptation of the SHOP algorithm. However, Pyhop uses recursion for task refinement, and the generated planning solutions do not preserve the hierarchy of the underlying task network. This lack of hierarchical information in the planning solution limits development of efficient replanning algorithms that could take advantage of the hierarchical nature of HTN plans. Here, we present an iterative, tree traversal-based

[1]An open-sourced version of Pyhop available at: `https://bitbucket.org/dananau/pyhop`

variant of Pyhop, namely IPyHOP, that preserves the hierarchy in the planning solution and provides users with a solution task network, or simply a solution tree, rather than a simple plan. Since IPyHOP iteratively generates a solution tree for task refinement, the structure of the solution tree represents the state of the planner. This representation allows developers to re-enter and continue planning from any desired planner state for the replanning process.

We also present an iterative hierarchical actor, RAE-lite, that uses task refinement to implement purely reactive acting. RAE-lite was inspired by Refinement Acting Engine (RAE)[2] [2, Chapter 3] and IPyHOP. RAE uses a hierarchical task-oriented operational representation with an expressive, general-purpose language offering rich control structures for closed-loop online decision-making. A collection of refinement methods describes alternative ways to handle tasks and react to events. Each method has a body that can be any complex algorithm. In addition to the usual programming constructs, the body may contain sub-tasks, which need to be refined recursively, and sensory-motor commands, which query and change the world non-deterministically. Additionally, RAE supports parallel refinement and execution of tasks. However, RAE-lite's methods for task refinement and execution are derived from IPyHOP; hence many functionalities offered by RAE's methods are limited/unavailable, and parallel task execution is impossible. However, RAE-lite is much simpler than RAE and can prove to be a better alternative for many practical scenarios.

Finally, we develop an efficient algorithm to integrate an HTN planner with

[2]An open-sourced version of RAE available at: `https://bitbucket.org/sunandita/rae`

an actor. A naive way of doing this would be to use repeated planning and replanning algorithms like Run-Lazy-Lookahead [2, Chapter 2]. However, this algorithm was initially designed for goal-based planners and did not take advantage of the hierarchical nature of HTN plans. We take inspiration from the execution of Refinement Acting Engines (RAE) [2, Chapter 3] and develop a repeated planning and replanning algorithm, Run-Lazy-Refineahead, that takes advantage of the hierarchical nature of HTN plans.

1.3 Thesis Organization

In chapter 2, the related work is discussed mainly concerning three areas: AI planning approaches, AI acting approaches, and systems that integrate AI planning with acting. In chapter 3, we begin by explaining the theory behind HTN planning and explain how it is implemented in Pyhop. We state some of its limitations and then explain how IPyHOP attempts to resolve them. We proceed by presenting the acting algorithm, RAE, and an HTN acting algorithm RAE-lite to describe a purely reactive HTN acting. We describe the Run-Lazy-Lookahead algorithm and its use in integrating HTN planning and acting or deliberative HTN acting. Based on the concepts developed so far, we present the Run-Lazy-Refineahead algorithm for deliberative HTN acting and conclude the chapter 3 by performing some analytical comparisons between Run-Lazy-Lookahead and Run-Lazy-Refineahead for deliberative HTN acting. In chapter 4 we describe an experimental domain for HTN planning. Furthermore, we discuss the design and setup of experiments to practically compare Run-Lazy-Lookahead and Run-Lazy-Refineahead algorithms for deliberative HTN acting. We provide the results and explain how the results support our analytic comparisons stated earlier. Finally, in chapter 5 we summarize our work and conclude the thesis. We also state the limitations of this work and provide some future research directions.

Chapter 2: Related Work

This chapter discusses the available scientific literature on different work areas related to deliberative acting. In section 2.1 we discuss various AI planning approaches relevant to this study. Then in section 2.2, we look at several approaches developed for AI acting. Finally, in section 2.3 we explore the numerous approaches developed for integrating AI planning and acting.

2.1 AI Planning

In the last decade, many commercial video games have used planners instead of classical behavior trees or finite-state machines to define agent behaviors. Planners allow looking ahead in time and prevent many problems of *purely reactive systems*. Goal-Oriented Action Planning (GOAP) [16] refers to a simplified Standford Research Institute Problem Solver (STRIPS) [17] -like planning architecture specifically designed for real-time control of autonomous character behavior in games. It is one of the earliest approaches to using an AI planner for a character's AI in a game. GOAP has continued to have a lasting impact within the video games industry. However, over time GOAP and its STRIPS-style approaches are now being adopted less frequently, with more contemporary titles adopting HTN planning. HTN planning is a widely adopted approach to AI planning in the gaming industry [18]. One of the first HTN planners was Nets of Action Hierarchies (NOAH) [19]. Since then, numerous HTN planners have been developed. Some of the best-known ones are Nonlin [20], System for Interactive Planning and Execution (SIPE) and SIPE-2 [21], Open Planning Architecture (O-Plan) [22] and its successor O-Plan2 [23], Universal Method Composition Planner (UMCP) [24], SHOP [12] and its successor SHOP2, and SHOP3 [13,14], and SIADEX [25]. Additionally, there are various HTN planners like Simple Hierarchical Planning Engine (SHPE) [26] that are specifically developed for AI planning in video games. A wide body of literature also exists on Monte Carlo tree search based planning. Monte Carlo tree search refers to simulated execution [27,28], sampling outcomes of action models [29,30], and hindsight optimization [31].

2.2 AI Acting

RAE [2, Chapter 3] is a popular acting algorithm, which in turn is inspired from Procedural Reasoning System (PRS) [32]. However, RAE and PRS are purely reactive systems. If they need to choose among several suitable *refinement methods* for a given task or event, they choose without trying to plan ahead. This lack of deliberation can lead to weird behavior of agents, where agents perform actions that will lead to their failure shortly. The purely reactive approach to acting has been extended with some planning capabilities in PropicePlan [33] and SeRPE [2, Chapter 3]. The basic idea in these approaches is to augment the acting procedure with predictive lookahead of the possible outcome of commands that can be chosen. This augmentation can be done, for example, by substituting the commands of actors with descriptive models of a planner. Various acting approaches similar to PRS and RAE have also been proposed [34–39]. Some with refinement capabilities and hierarchical models [40–42]. While such systems offer expressive acting environments, e.g., with real-time handling primitives, most do not perform reasoning about alternative refinements.

2.3 Integrating AI planning and acting

In [43] authors 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 [2, Chapter 2]. Run-Concurrent-Lookahead is a procedure in which the acting and planning processes run concurrently. Each time an action is performed, the action comes from the most recent plan that Lookahead has provided. Other similar algorithms like Run-Lookahead and Run-Lazy-Lookahead are also defined in [2, Chapter 2]. Here Lookahead is any *online* planning algorithm in the mentioned procedures. The online nature of a planner means that at a given instance, the plan returned might not guarantee to solve the planning problem; however, it has to provide at least a *partial* solution. 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.

Much work has been done in robotics to integrate planning and acting. In [44] show how HTN planning can be used in robotics. In [45] authors describe the integration of task and motion planning using an HTN approach. Motion primitives are assessed with a specific solver through sampling for cost and feasibility. An algorithm called SAHTN extends the usual HTN search with a bookkeeping mechanism to cache previously computed motions. In comparison to this work, our approach does not integrate specific constructs for motion planning. However, it is more generic regarding the integration of HTN planning and acting.

Chapter 3: Planning and Acting Algorithms

This chapter first defines the HTN planning formulation in section 3.1. Some of the essential terms and representations are defined, and an abstract HTN planning algorithm is stated. In section 3.2 we present some details about the planner Pyhop and summarize its HTN planning algorithm. Some limitations of Pyhop are recognized, and the planner IPyHOP is presented as a solution in section 3.3. The crucial differences between Pyhop and IPyHOP are pointed out, and the modified algorithm is summarized. In section 3.4 we describe a purely reactive HTN actor, RAE-lite, that was derived from IPyHOP's HTN refinement algorithm and RAE's style of task refinement and execution. Finally, in section 3.5 we describe ways of integrating an HTN planner with an actor. We describe the Run-Lazy-Lookahead algorithm and its use in deliberative HTN acting. And then, based on the concepts developed so far, we present the Run-Lazy-Refineahead algorithm for deliberative HTN acting. We conclude this chapter by performing some analytical comparisons between Run-Lazy-Lookahead and Run-Lazy-Refineahead for deliberative HTN acting.

3.1 HTN Planning Formulation

We follow the HTN formulation as described in [1, Chapter 11]. However, we reiterate some key concepts from their HTN formulation in this section for preciseness and completeness. Understanding these concepts is vital to understanding the subsequent discussions in this thesis.

We borrow the definitions of terms, literals, operators, actions, and plans from classical planning. The plan is usually represented as the symbol π throughout this text. The definition of the prediction function $\gamma(s, a)$, which tells the result of applying an action a to a state s, is also the same as in classical planning. However, the language also includes tasks, methods, and task networks used in defining a planning problem and its solutions.

One new kind of symbol is a *task symbol.* Every operator symbol is also a task symbol, and there are some additional task symbols called *non-primitive task symbols.* A task is an expression of the form $t(r_1, ..., r_k)$ such that t is a task symbol, and $r_1, ..., r_k$ are terms. If t is an operator symbol, then the task is called a *primitive task*; otherwise, the task is called a *non-primitive task.* In classical planning, a literal is *ground* if it contains no variables, and *unground* otherwise. Therefore, a task is ground if all of the terms are ground; otherwise, it is unground. Ground primitive tasks are accomplished by using an action.

An action a is a 3-tuple

$$a = (\mathit{name}(a), \mathit{precond}(a), \mathit{effects}(a))$$

which accomplishes a ground primitive task t in a state s if $\mathit{name}(a) = t$ and a is applicable to s. The state should satisfy the $\mathit{precond}(a)$ before the action is executed and the action modifies the state so that it satisfies the $\mathit{effects}(a)$ after its execution.

A *task network* is a pair

$$w = (U, C)$$

where U is a set of task nodes and C is a set of constraints. Each constraint in C specifies a requirement that must be satisfied by every plan that is a solution to a planning problem.

An HTN *method* is a 4-tuple

$$m = (\mathit{name}(m), \mathit{task}(m), \mathit{subtasks}(m), \mathit{constr}(m))$$

in which the elements are described as follows.

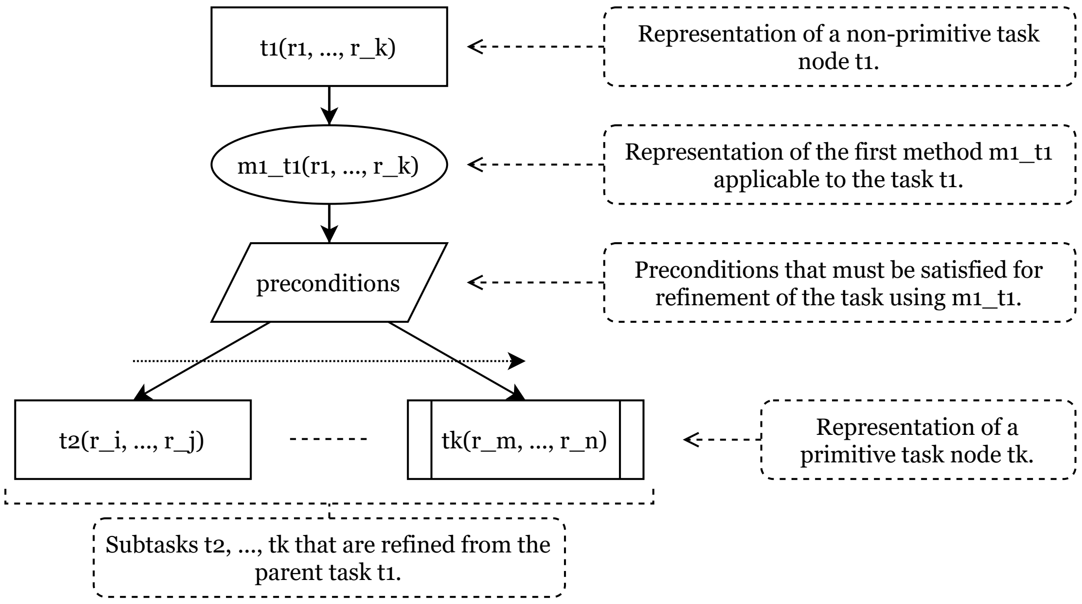
- $\mathit{name}(m)$ is an expression of the form $n(x_1, ..., x_k)$, where n is a unique method symbol (i.e., no two methods in the planning domain can have the same method symbol), and $x_1, ..., x_k$ are all of the variable symbols that occur anywhere in m.

- $\mathit{task}(m)$ is a non-primitive task.

- $(\mathit{subtasks}(m), \mathit{constr}(m))$ is a task network.

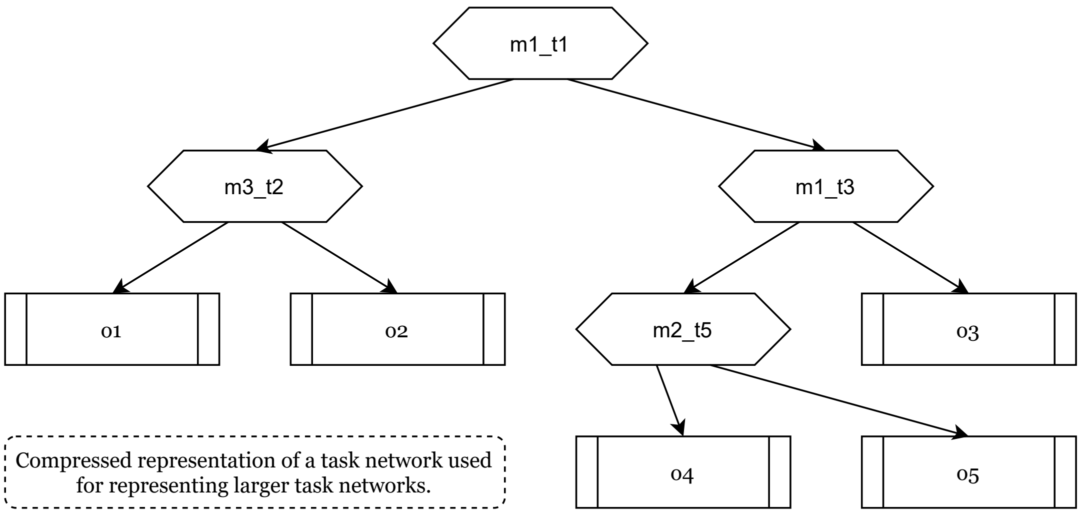
A task can have multiple methods where each method defines a possible way of refining that task.

Figure 3.2 visualizes the above formulated representation of a task network.



t1(r1, ..., r_k)
Representation of a non-primitive task node t1.
m1_t1(r1, ..., r_k)
Representation of the first method m1_t1 applicable to the task t1.
preconditions
Preconditions that must be satisfied for refinement of the task using m1_t1.
t2(r_i, ..., r_j)
tk(r_m, ..., r_n)
Representation of a primitive task node tk.
Subtasks t2, ..., tk that are refined from the parent task t1.

(a)



m1_t1
m3_t2
m1_t3
o1
o2
m2_t5
o3
o4
o5
Compressed representation of a task network used for representing larger task networks.

(b)

Figure 3.1: Task network visualizations: (a) Visualization of a task, method, preconditions, sub-tasks, and operators. (b) Compressed visualization of a task network used for visualizing larger networks.

```
Abstract-HTN($s, U, C, O, M$):
if $(U, C)$ can be shown to have no solution then
    return failure
else if $U$ is primitive then
    if $(U, C)$ has a solution then
        nondeterministically let $\pi$ be any such solution
        return $\pi$
    else
        return failure
else
    choose a primitive task node $u \in U$
    active $\leftarrow \{m \in M \mid task(m)$ is unifiable with $t_u\}$
    if active $\neq \emptyset$ then
        nondeterministically choose any $m \in active$
        $\sigma \leftarrow$ an mgu for $m$ and $t_u$ that renames all variables of $m$
        $(U', C') \leftarrow \delta(\sigma(U, C), \sigma(u), \sigma(m))$
        $(U', C') \leftarrow$ apply-critic$(U', C')$        \\ this line is optional
        return Abstract-HTN$(s, U', C', O, M)$
    else
        return failure
```

Algorithm 1: The Abstract-HTN procedure. [1, Chapter 11]

Suppose that $w = (U, C)$ is a task network, $u \in U$ is a task node, t_u is its task, m is an instance of a method in M, and $task(m) = t_u$. Then m decomposes u into $subtasks(m')$, producing the task network

$$\delta(w, u, m) = ((U - u) \cup subtasks(m'), C' \cup constr(m')),$$

where C' is a modified version of C.

An HTN *planning domain* is a pair

$$\mathcal{D} = (O, M)$$

And an HTN *planning problem* is a 4-tuple

$$\mathcal{P} = (s_0, w, O, M),$$

where s_0 i sthe initial state, w is the inital task network, O is a set of operators, and M is a set of HTN methods.

HTN planning procedures must both instantiate operators and decompose tasks. Because there are several different ways to do both of these things, the number of different possible HTN planning procedures is quite large. Abstract-HTN shown in algorithm 1 is an abstract procedure that includes many (but not necessarily all) of them.

3.2 HTN Planning in Pyhop

Pyhop is a domain-independent HTN planning system written in Python. Pyhop plans for tasks 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 Pyhop knows the complete world-state at each step of the planning process, it can use highly expressive domain representations. The planning algorithm is like the one in SHOP, but with several differences that make it easier to integrate it with ordinary computer programs:

- Pyhop represents states of the world using ordinary variable bindings, not logical propositions. A state is just a Python object that contains the variable bindings. For example, you might write $s.loc['v'] =' d'$ to say that vehicle v is at location d in state s.

- To write HTN operators and methods for Pyhop, you do not need to learn a specialized planning language. Instead, you write them as ordinary Python functions. The current state (e.g., s in the above example) is passed to them as an argument.

If the initial tasks to be planned are totally ordered, we can sometimes dispense with the graph notation for the task network $w = (U, C)$, and instead write w as the sequence of tasks, namely, $w = \langle t_1, ..., t_k \rangle$ where t_1 is the task in the first node of U, t_2 is the task in the second node of U, and so forth. Pyhop uses this convention for defining its tasks. The HTN planning algorithm used by Pyhop is presented in algorithm 2.

```
1  **Pyhop**($s, w, \pi, O, M$):
2  if $w$ is empty then
3      return $\pi$
4  $t \leftarrow$ first task in $w$
5  if $t$ is primitive then
6      $s' \leftarrow t(s, r_1, ..., r_k)$
7      if $s'$ is valid then
8          $w' \leftarrow w \backslash t$
9          $\pi' \leftarrow \pi \cup t$
10         solution $\leftarrow$ **Pyhop**($s', w', \pi', O, M$)
11         if solution is valid then
12             return solution
13 if $t$ is non-primitive then
14     foreach $m \in \{$methods relevant to $t\}$ do
15         $t' \leftarrow m(s, r_1, ..., r_k)$
16         if $t'$ is valid then
17             $w' \leftarrow t' \cup w \backslash t$
18             solution $\leftarrow$ **Pyhop**($s, w', \pi, O, M$)
19             if solution is valid then
20                 return solution
21 return failure
```

Algorithm 2: HTN Planning in Pyhop.

It can be observed that Pyhop uses recursion for task refinement. Writing the algorithm as a recursive algorithm is intuitive, and it follows the Abstract-HTN algorithm described in algorithm 1. The algorithm is also simple to implement, and the recursion stack efficiently handles the refinement and backtracking during task planning. However, in practice, this limits the level of control the user has over the refinement process.

Also, tasks are represented as a sequence of tasks rather than a task network, and the generated planning solution π is a sequence of primitive tasks. Even though the recursion stack of the algorithm implicitly represents the task network, explicit representation as an acyclic digraph is not available to the user. Thus the hierarchical representation of the underlying refined task network is lost to the user.

3.3 HTN Planning in IPyHOP

IPyHOP is very similar to Pyhop with two key differences:

- IPyHOP uses an iterative tree traversal/generation routine for task refinement. However, writing the HTN planning algorithm as an iterative algorithm is more complicated and less intuitive than writing it as a recursive algorithm. We also need to define the refinement and backtracking as tree traversal algorithms explicitly. However, it provides immense control to the user over how the algorithm executes the refinement process.

- Even though the tasks are totally ordered, they are represented as a task network rather than a sequence of tasks. The task network is represented as an acyclic digraph as initially suggested in the HTN planner formulation in section 3.1. This representation also means that the plans are represented as a solution tree, a refined task network, rather than a sequence of primitive tasks.

Deriving from the formulation stated in section 3.1, let u represent a grounded task node. Then,

- $\mathit{task}(u)$ defines the grounded task $t = t(r_1, ..., r_k)$ corresponding to u.
- $\mathit{refined}(u) \in \{\mathit{true}, \mathit{false}\}$ 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.
- $\mathit{visited}(u) \in \{\mathit{true}, \mathit{false}\}$ represents if the node has been visited.
- $\mathit{state}(u)$ represents the state when the node was first visited.
- $\mathit{methods}(u)$ represents the methods applicable to the task t that haven't been used for refinement of u, given that the task is non-primitive.

The HTN planning algorithm used by IPyHOP is presented in algorithm 3. backtrack(w, u) is a subroutine that modifies the task network given that refinement of node u failed. After backtracking, the non-primitive task node u', the node refined before the current task node u, is again marked for refinement. The backtracking algorithm is described in algorithm 4. add_nodes(u, t') adds the sub-tasks t' as nodes to the refined node u.

```
IPyHOP($s, w, O, M$):
$p \leftarrow root(w)$
while $true$ do
    $u \leftarrow$ first unrefined node in BFS_Successors($p$)
    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
            $s \leftarrow s'$
            $refined(u) \leftarrow$ true
        else
            $w, u \leftarrow$ 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
            $w, u \leftarrow$ backtrack($w, u$)
            $p \leftarrow parent(u)$
return $w$
```

Algorithm 3: HTN Planning in IPyHOP.

```
1  **backtrack**$(w, u)$:
2  $p \leftarrow parent(u)$
3  $W_p \leftarrow$ Preorder_DFS$(p)$
4  **foreach** $v \in reversed(W_p)$ **do**
5      $refined(v) \leftarrow false$
6      **if** $v$ *is non-primitive* **then**
7          $W_v \leftarrow descendants(v)$
8          $w \leftarrow w \backslash W_v$
9          return $w, v$
10 $w \leftarrow \{root(w)\}$
11 return $w, root(w)$
```

Algorithm 4: IPyHOP Backtracking.

3.4 RAE-lite - A purely reactive HTN Actor

RAE-lite is a purely reactive HTN actor that takes its inspiration from RAE [2, Chapter 3] and IPyHOP. RAE uses a library of methods M to address new tasks the actor has to perform and new events it has to react to. The input to RAE consists of (i) a set of facts reflecting the current state of the world ξ , (ii) a set of tasks to be performed, and (iii) a set of events corresponding to exogenous occurrences to which the actor may have to react. These three sets change continually. Tasks come from task definition sources, for example, a planner or a user. Events come from the execution platform, for example, through a sensing and event recognition system. Facts come either from the execution platform, as updates of the perceived state of the world, or from RAE, as updates of its reasoning state. RAE is a well-defined acting algorithm capable of handling the refinement of multiple tasks at the same time. However, this makes RAE quite challenging to implement.

In section 3.3 we discussed an iterative graph traversal based algorithm for

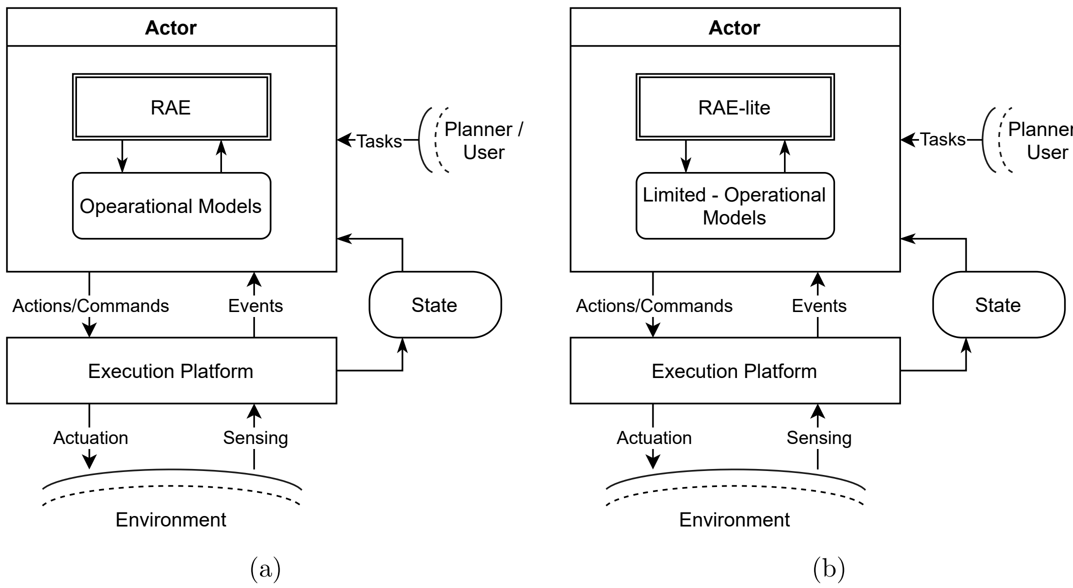


Figure 3.2: Actor architectures: (a) Interaction of RAE with execution platform. (b) Interaction of RAE-lite with execution platform.

HTN planning. In this section, we will discuss an iterative graph traversal-based algorithm for HTN acting. RAE's task methods use a hierarchical task-oriented operational representation with an expressive, general-purpose language offering rich control structures for closed-loop online decision-making. RAE-lite, on the other hand, uses task methods similar to the ones used in HTN planners for task refinement of non-primitive tasks. You are allowed to have any general-purpose code in these methods; however, the task refinement function of these methods follows that of HTN methods. Furthermore, like RAE, RAE-lite does not use descriptive models for operators. Operators can be any general-purpose code or a command to the execution platform. The interaction between RAE and execution platform is illustrated in figure 3.2(a). The interaction between RAE-lite and execution platform is illustrated in figure 3.2(b).

The HTN acting algorithm used by RAE-lite is presented in algorithm 5.

```
RAE-lite($s, w, M$):
$p \leftarrow root(w)$
while $true$ do
    $u \leftarrow$ first unrefined node in BFS_Successors($p$)
    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
        send-request-to-execution-platform($request\text{-}type, w, u$)
        $\{s, w, status, events\} \leftarrow$ execution-platform-response()
        $refined(u) \leftarrow true$
        handle-events($events$)
        if $status = failure$ then
            $w, u \leftarrow$ retry-parent($w, u$)
            $p \leftarrow parent(u)$
    if $t$ is non-primitive then
        foreach $m \in methods(u)$ where $m \in M$ do
            $\{s, t'\} \leftarrow m(s, r_1, ..., r_k)$
            $methods(u) \leftarrow methods(u) \backslash m$
            if $t'$ is valid then
                $refined(u) \leftarrow true$
                add_nodes($u, t'$)
                $p \leftarrow u$
                break;
        if not $refined(u)$ then
            $w, u \leftarrow$ retry-parent($w, u$)
            $p \leftarrow parent(u)$
return $w$
```
Algorithm 5: HTN Acting in RAE-lite.

```
retry-parent($w, u$):
$p \leftarrow parent(u)$
$refined(p) \leftarrow false$
$W_p \leftarrow descendants(p)$
$refined(W_p) \leftarrow invalid$
return $w, p$
```
Algorithm 6: RAE-lite retry-parent algorithm.

retry-parent(w, u) is a subroutine that modifies the pointer to the current task in the task network given that the execution of node u failed. After retry-parent, the non-primitive task node u', the node which is the parent of the current task node u, is again marked for refinement. The retry-parent algorithm is described in algorithm 6. add_nodes(u, t') adds the sub-tasks t' as nodes to the refined node u.

3.5 Integrating IPyHOP with an Actor

As explained in section 2.3, a popular way of integrating a planner and an actor is by using algorithms like Run-Lazy-Lookahead. In sub-section 3.5.1 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 subsection 3.5.2 we describe the Run-Lazy-Refineahead algorithm for deliberative HTN acting. We also provide an analytic comparison between Run-Lazy-Lookahead and Run-Lazy-Refineahead for deliberative HTN acting.

3.5.1 Run-Lazy-Lookahead

Algorithm 7 presents the Run-Lazy-Lookahead algorithm as presented in [2, Chapter 2]. (Σ, s, g) is a planning problem, and Lookahead is an online planning algorithm.

Run-Lazy-Lookahead 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 computation-

```
Run-Lazy-Lookahead($\Sigma, g$):
$s \leftarrow$ abstraction of observed state $\xi$
while $s \not\models g$ do
    $\pi \leftarrow$ Lookahead($\Sigma, s, g$)
    if $\pi = failure$ then
        return $failure$
    while $\pi \neq \langle\rangle$ and $s \not\models g$ and $Simulate(\Sigma, s, g, \pi) \neq failure$ do
        $a \leftarrow$ pop-first-action($\pi$)
        perform($a$)
        $s \leftarrow$ abstration of observed state $\xi$
```

Algorithm 7: Run-Lazy-Lookahead [2, Chapter 2]

ally expensive to call Lookahead, 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, 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

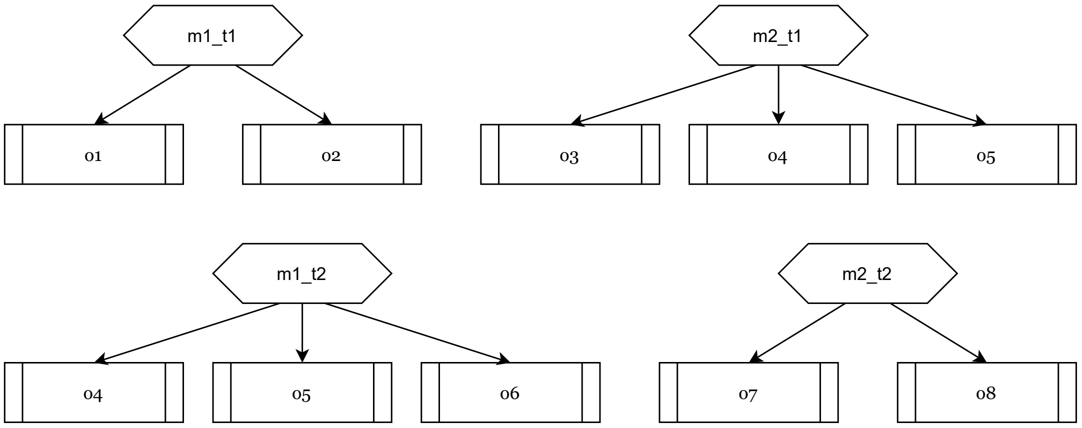


Figure 3.3: Refinement of task $t1$ using $m1\_t1$ and $m2\_t1$. And refinement of task $t2$ using $m1\_t2$ and $m2\_t2$. (Example 1).

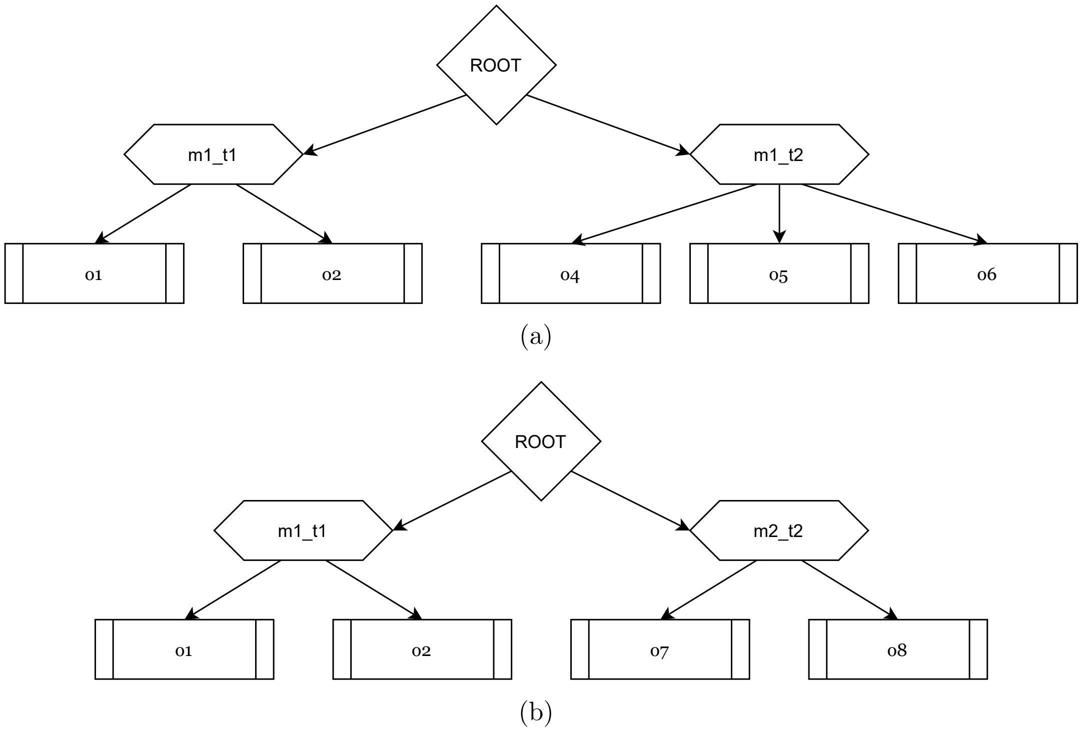


Figure 3.4: Task network visualizations: (a) After first planning attempt. (b) Re-planning after failure in execution of o6.

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 operators defined here are grounded. These individual refinements can be visualized in figure 3.3. In this example, for the sake of simplicity, lets assume that all the methods and operators have no pre-conditions, and all are applicable 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 3.4(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, $o6$ non-deterministically 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 3.4(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 weird 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 intractable. One of the most significant limitations of HTN planning is the enormous 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.

3.5.2 Run-Lazy-Refineahead

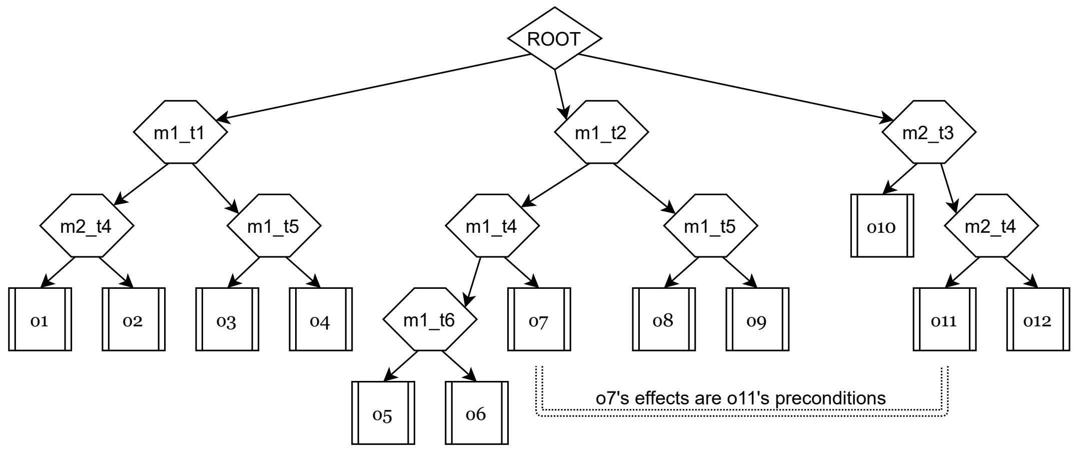
The problems explained in sub-section 3.5.1 mainly happen because of the incompatibilities between the definition of the Lookahead planner and the definition of an HTN planner. The signature of the Lookahead planner is (Σ, s, g), whereas the signature of HTN planners is (s, w, O, M). Here Σ is the planning domain, which is modeled by $\{O, M\}$ for an HTN planner. 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.

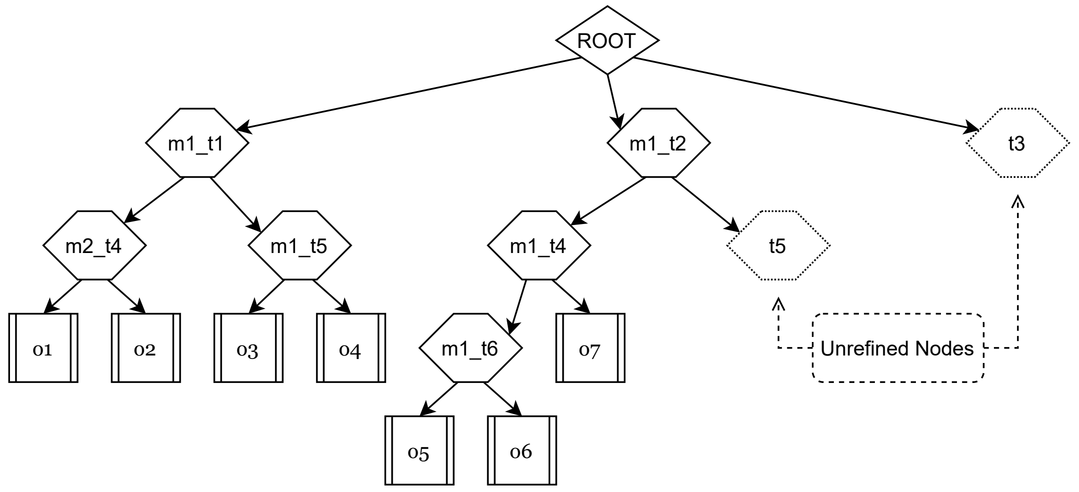
The task network described in example 1 is simplistic, and finding the modified task network is trivial. We compute the parent task node of the failed primitive task node and only re-plan for the computed task node. However, for a more complicated task network, this will not work. We will have to come 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 3.5(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 $\pi = \langle o1, o2, ..., o10, o12 \rangle$. However, while executing this plan, assume that $o7$ non-deterministically 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.

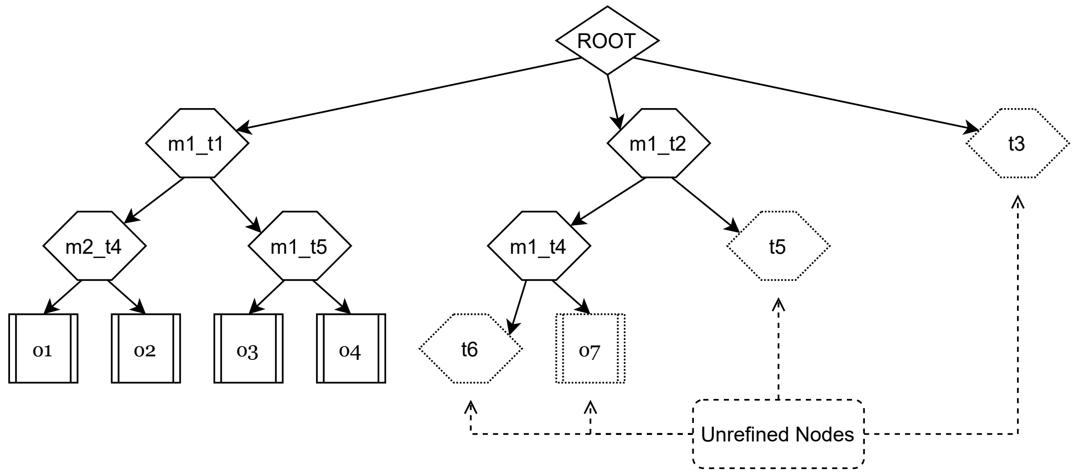
In the above explained scenario, we should modify the solution task network by removing refinements of all the tasks that come after the failed node $o7$ in the Pre-ordered DFS traversal. Alternatively, this could be done more efficiently using the



(a)



(b)



(c)

Figure 3.5: (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).

algorithm Un-Refine-Post as described in algorithm 8. At this point, the modified task network should look like figure 3.5(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 3.5(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. ■

```
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
            $W_v \leftarrow descendants(v)$
            $w \leftarrow w \setminus W_v$
    $u \leftarrow p$
    if $u = root(w)$ then
        break
return $w$
```

Algorithm 8: Un-Refine-Post. Algorithm used to modify a task network w after failure at u.

This way of repeated planning and acting leads to the formulation of the Run-Lazy-Refineahead algorithm described in algorithm 9.

Run-Lazy-Refineahead is a repeated planning and acting algorithm for integrating HTN planning and acting. Here, Refineahead is any online HTN planner

```
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
```
Algorithm 9: Run-Lazy-Refineahead.

that provides the solution as a refined task network and provides control over its backtracking feature.

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 8, and backtracking is performed using the Backtrack algorithm of an HTN planner, ex. algorithm 4. 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.

Chapter 4: Experimental Evaluation

This chapter describes the experimental setup and evaluation of the two deliberative HTN acting algorithms: (i) Run-Lazy-Lookahead and (ii) Run-Lazy-Refineahead. We use IPyHOP as the HTN planner for both of these approaches. In section 4.1 we explain the domain we used to run our experiments. In section 4.2 we explain the design and setup of experiments and explain the metrics we use to compare the two approaches. In section 4.3 we provide and explain the results obtained with our experiment, and in section 4.4 we provide the summary of this chapter.

4.1 RoboSub Domain

Every year RoboNation, Inc.[1] hosts the RoboSub challenge[2] [3] to provide various teams consisting of high school, undergraduate, and graduate students with a platform to demonstrate their engineering acumen in designing Autonomous Underwater Vehicles (AUVs). RoboSub is an international student competition. Student teams from around the world design and build Robotic submarines, or in other

[1]RoboNation official website: `https://robonation.org`
[2]RoboNation's page on RoboSub: `https://robonation.org/programs/robosub`
[3]RoboSub official website: `https://robosub.org`

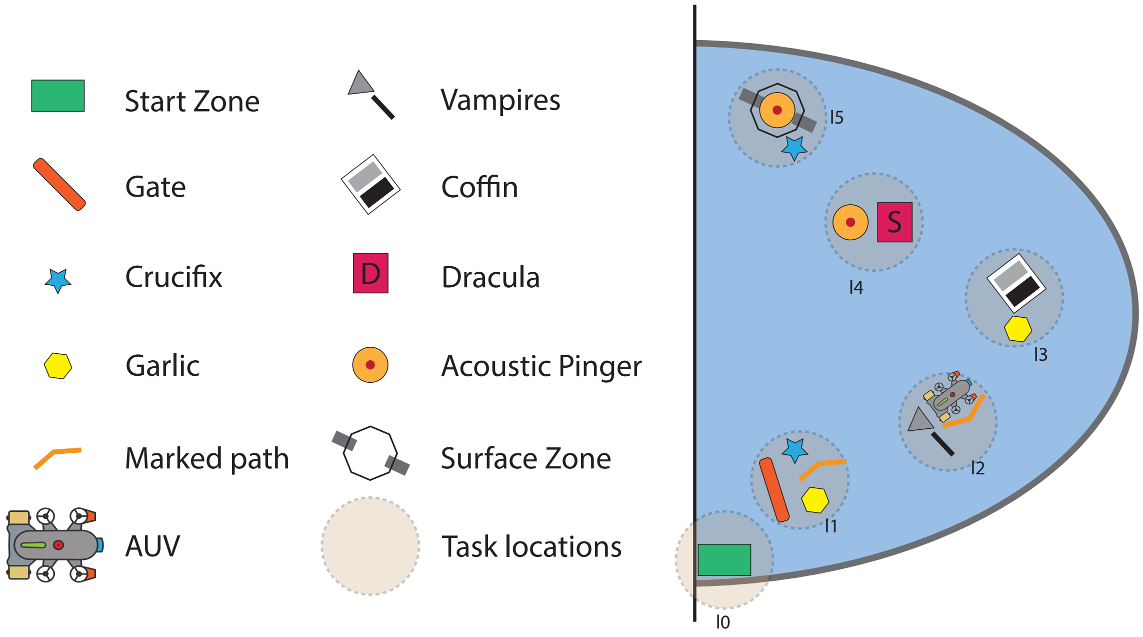


Figure 4.1: An illustration of a sample planning problem for the RoboSub 2019 competition.

words, AUVs. The behaviors demonstrated by these experimental AUVs mimics those of real-world systems currently deployed around the world for underwater exploration, seafloor mapping, and sonar localization, amongst many others. The fundamental goal of the RoboSub competition is for an AUV to demonstrate its autonomy by completing underwater tasks, with a new theme each year. Robotics @ Maryland[4] is a student-run robotics team at the University of Maryland, College Park that participates in RoboSub competition every year. The domain used for these experiments is a modified version of the domain we used for designing the autonomous planning functionality of Qubo[5], our entry to the RoboSub 2019[6] competition.

The theme of the RoboSub 2019 competition was based on an imaginary un-

[4]Robotics @ Maryland official website: `http://ram.umd.edu`
[5]Webpage describing Qubo AUV: `http://ram.umd.edu/html/qubo.html`
[6]Details about RoboSub 2019 competition: `https://robosub.org/programs/2019`

derwater realm where our AUV had to slay Vampires and Dracula. Figure 4.1 illustrates a sample planning problem for the competition. The fundamental goal of the mission was for an AUV to demonstrate its autonomy by interacting with various vampires. Orange guide markers were placed to help direct the vehicle to the beginning tasks, and two acoustic pingers were set up to guide the AUV to the remaining two tasks. Along the way, garlic markers and crucifix markers were placed, which the vehicle could pick up and use in other locations. The AUVs had to complete the following tasks:

- Enter the Undead Realm (Gate): The AUV had to pass a validation gate. The validation gate had a separator that separated it into 40% and 60% sections. Higher points were awarded for AUVs that crossed the gate from the 40% section compared to those that crossed it from the 60% section.

- Pickup Garlic and Crucifix (Markers): Two kinds of objects were placed throughout the course. (i) garlic markers and (ii) crucifix markers. The AUVs were supposed to pick them up for use in later tasks of the competition.

- Recognize and Trace Path (Marked Paths): Orange path markers were placed to direct the AUVs to different task zones. The AUVs were expected to trace the marked paths and use them to find the next task zones.

- Slay Vampires (Touch buoys): There were two "buoys" that were moored to the floor at two places in the course. One buoy was two-sided with images of a Jiangshi on both sides. In China, a Jiangshi (hopping vampire) is a deceased loved one brought back home by a sorcerer. Furthermore, the other buoy was three-sided with images of Aswang, Draugr, and Vetalas. The Aswang are creatures from the Philippines, Malaysia, Cambodia, and Indonesia that take a form of an attractive girl by the day and develop wings and a long, hollow, thread-like tongue by night. The Draugr is an Icelandic parasitic ghost who roams the earth and harasses the living to drive them mad or even kill them. In India, the Vetalas are undead ghoul-like beings that inhabit corpses, hang upside down on trees, found on cremation ground and cemeteries. The task was to touch these buoys with the AUV. Higher points were allotted if the AUV touched the side of the buoy facing away from the gate.

- Drop Garlic (Drop markers): For this task, the AUVs were supposed to drop the earlier picked garlic markers into a bin. The bin is initially closed, and the task of opening the bin and dropping the garlic markers in an opened bin yielded more points than dropping the garlic markers in closed bins.

- Stake Through Heart (Manipulation/torpedoes): To reach the location zone of this task, the AUVs first had to sense and localize an acoustic pinger. The task was inspired by the lore of the Romanian Count Dracula. Dracula can be killed by driving a wooden stake through the heart and beheading. To complete this task with the maximum points, the AUV had to first decapitate Dracula by pulling a lever and then shoot two torpedoes through holes on the task board.

- Expose to Sunlight (Retrieve object(s), surface, move/release objects(s)): The final task in the competition was to surface the AUV with a crucifix marker at a surface zone. The surface zone could be found by localizing another acoustic pinger placed in the course.

Completion of some of these tasks was compulsory, while others could be skipped. 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.

4.2 Experimental Setup

We wanted to statistically analyze the performance of Run-Lazy-Lookahead and Run-Lazy-Refineahead approaches for deliberative HTN acting for the above-defined tasks. Let X denote the set containing all possible states. Based on our calculations, the state space containes approximately $1.3318E + 19$ unique states, i.e. $\|X\| = 1.3318E + 19$. We calculate our state space based on possible values of each state variable in the state. For the RoboSub competition, the initial location of the robot was fixed, and a few other constraints were specified. However, the location of various objects in the planning problem was varied. Let I denote the set containing all possible values of the initial states, where $I \subset X$. Based on our calculations, approximately $2.0213E+08$ unique states could be the initial state, i.e. $\|I\| = 2.0213E+08$. We wanted to solve the planning problem for the competition's tasks, given that our initial state is randomly sampled from the initial state space I. Let the sampled set be denoted as S, where $S \subset I$. We sampled $1E + 4$ initial states randomly and uniformly from I, i.e. $\|S\| = 1E + 4$.

A state s, where $s \in S$, is chosen as the starting state for the *planning problem*, p. 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 non-deterministic, 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*, t. A test case t_i corresponds to the i^{th} planning problem with the initial state s_i, where s_i is the i^{th} state in S. The test case results are stored, and we repeat this deliberative HTN acting process for all the states in S. The execution of all the test cases t_i is known as an *experiment*, e, where $e = \langle t_1, t_2, ..., t_j \rangle$, where $j = \|S\|$.

Since the execution environment is non-deterministic, multiple test case executions lead to different results for the calculated metrics. This difference in metrics means that the results obtained from an experiment e_1 will differ from the results obtained from another experiment e_2. To have a reasonable estimate of our metrics, we repeat the experiment 11 times, i.e. $E = \langle e_1, e_2, ..., e_{11} \rangle$.

For our comparison, we calculate the following five metrics:

- Total nodes expanded: This metric calculates the total number of task nodes expanded/refined by the planner in the given test case.

- Total actions planned: This metric calculates the total number of primitive task nodes processed by the planner in the given test case. The primitive task nodes are the leaf nodes of the solution task network. This metric is a measure of the total plan length for a test case.

- 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.

4.3 Results

The raw data collected d_{raw} in each experiment $e \in E$ described in section 4.2 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 $[e \times a \times t \times m]$. Here $e = 11$ is the number of experiments performed, $a = 2$ is the number of deliberative HTN acting algorithms being compared, $t = 1E + 4$ is the number of test cases solved, and $m = 5$ is the number of metrics evaluated.

The $D_{results}$ dataset was processed further by performing a reduce mean operation across the $zero^{th}$ axis of the dataset. Thus the dataset D_{exp_mean} of size $[a \times t \times m]$ 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 4.2(a), 4.3(a), 4.4(a), 4.5(a), and 4.6(a).

Another form of post-processing was done on D_{raw} to generate the $D_{equivalent}$ dataset. As stated earlier, the most significant reason for variance in values of a calculated metric for a given test case across experiments is due to the non-determinism of the execution environment. This non-determinism causes the same test case to have a varying number and location of failures across experiments. For example, for a test case t_i in experiment e_j, $f(t_i, e_j)$ failures occurred during the deliberative acting process, and for the same test case t_i in a different experiment e_k, $f(t_i, e_k)$ failures occurred. Here $f(t_i, e_j)$ might or might not be equal to $f(t_i, e_k)$. This makes the calculation of relation of metrics for different algorithms difficult. To alleviate this problem, we generate a modified dataset $D_{equivalent}$ of size $[a \times t \times m]$, where the data corresponding to each test case t_i for both the algorithms featured the same number of failures. The relation of metrics calculated for different algorithms in dataset $D_{equivalent}$ are illustrated in figures 4.2(b), 4.3(b), 4.4(b), 4.5(b), and 4.6(b).

Note that the metrics represented by the $D_{equivalent}$ dataset are not very 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 $D_{equivalent}$ dataset, we will need to perform more experiments such that multiple data points are available for each metric in the dataset.

Figure 4.2(a) shows the distribution of the metric - Total nodes expanded, across the different test cases as a histogram plot. The relation of this metric for the two deliberative acting algorithms is visualized in figure 4.2(b) as a scatter plot.

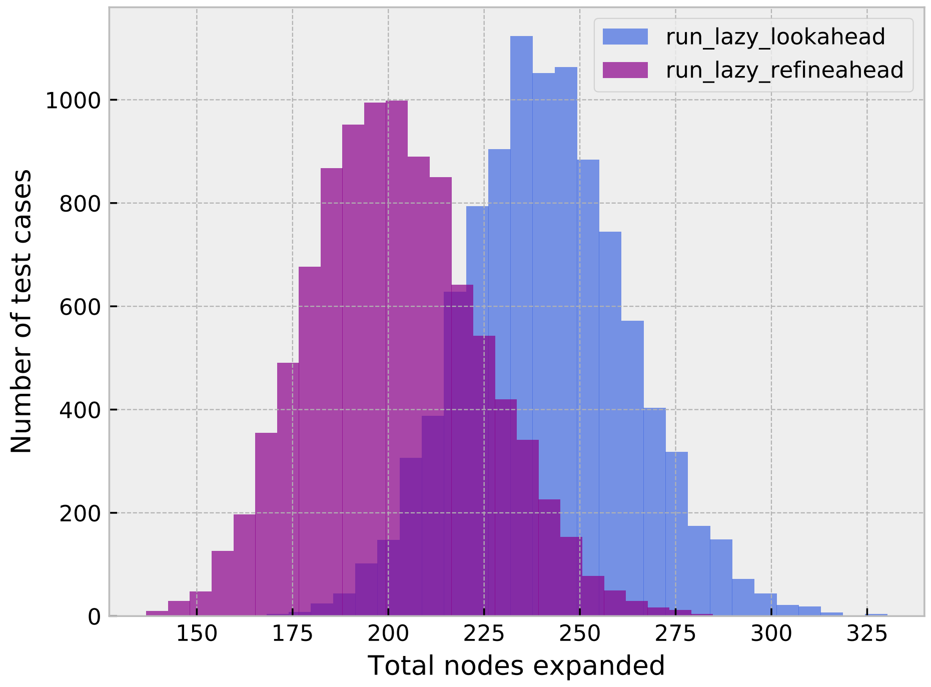
Figure 4.3(a) shows the distribution of the metric - Total actions planned, across the different test cases as a histogram plot. The relation of this metric for the two deliberative acting algorithms is visualized in figure 4.3(b) as a scatter plot.

Figure 4.4(a) shows the distribution of the metric - Total iterations taken, across the different test cases as a histogram plot. The relation of this metric for the two deliberative acting algorithms is visualized in figure 4.4(b) as a scatter plot.

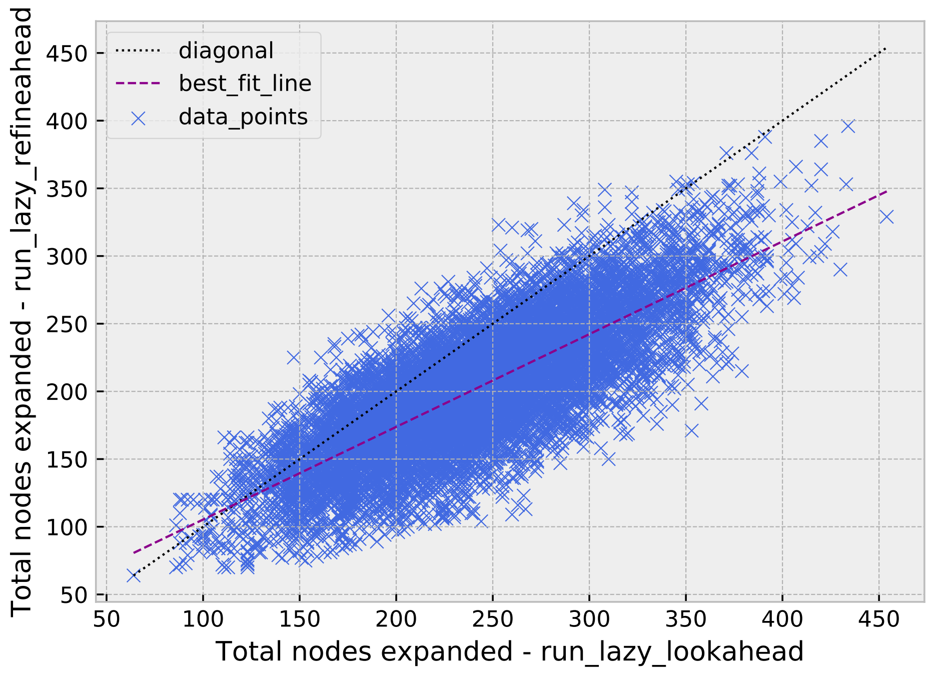
Figure 4.5(a) shows the distribution of the metric - Total action cost, across the different test cases as a histogram plot. The relation of this metric for the two deliberative acting algorithms is visualized in figure 4.5(b) as a scatter plot.

Figure 4.6(a) shows the distribution of the metric - Final state reward, across the different test cases as a histogram plot. The relation of this metric for the two deliberative acting algorithms is visualized in figure 4.5(b) as a scatter plot.

The numerical values corresponding to these results are presented in tables 4.1

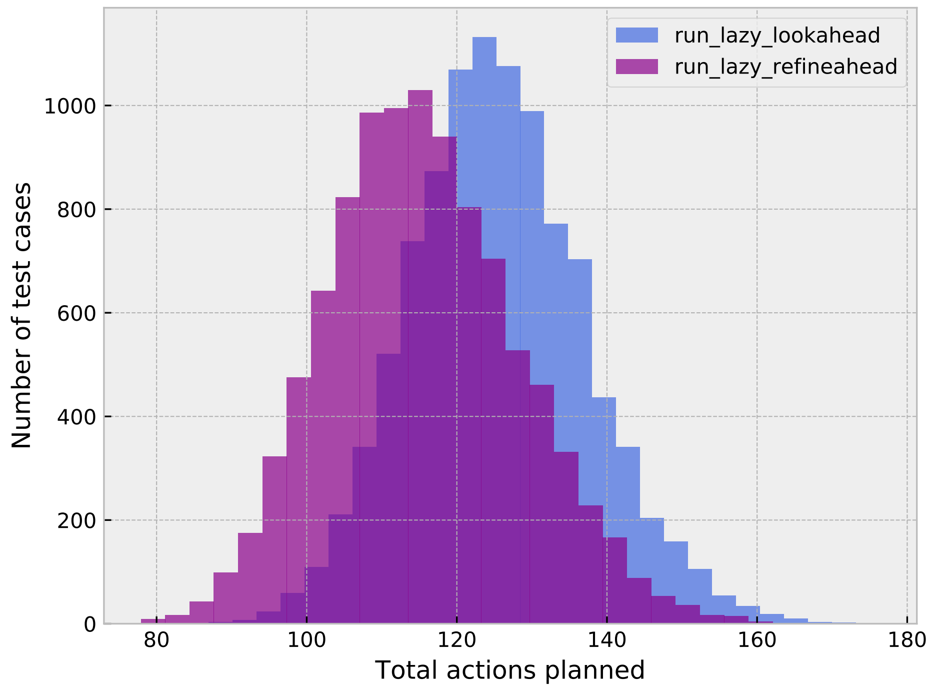


(a)

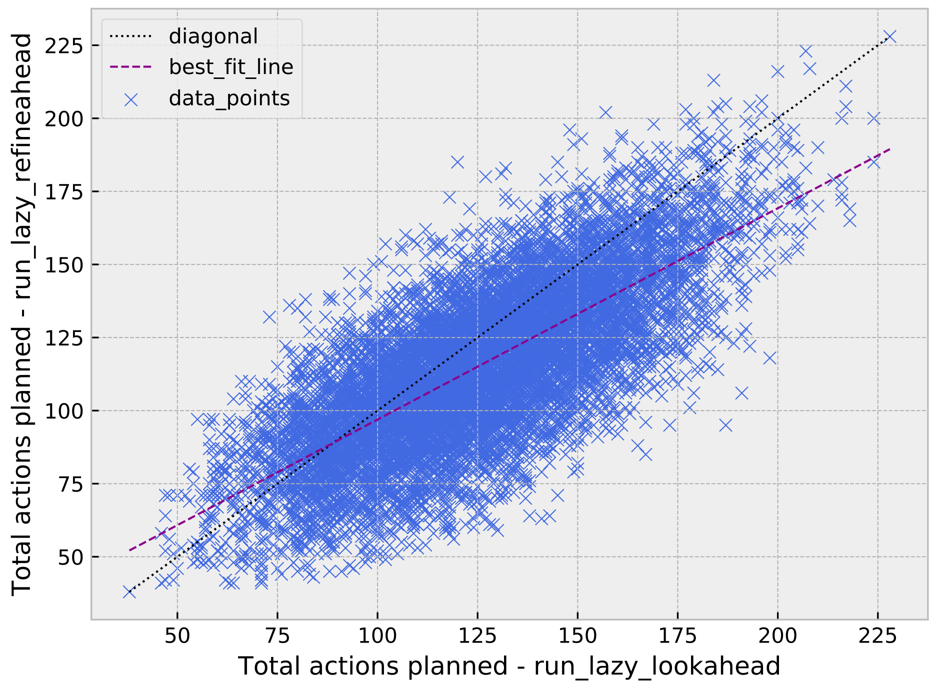


(b)

Figure 4.2: Results of metric - Total nodes expanded (a) Distribution visualized using histograms (b) Relation visualized by fitting a line on the scatter plot.

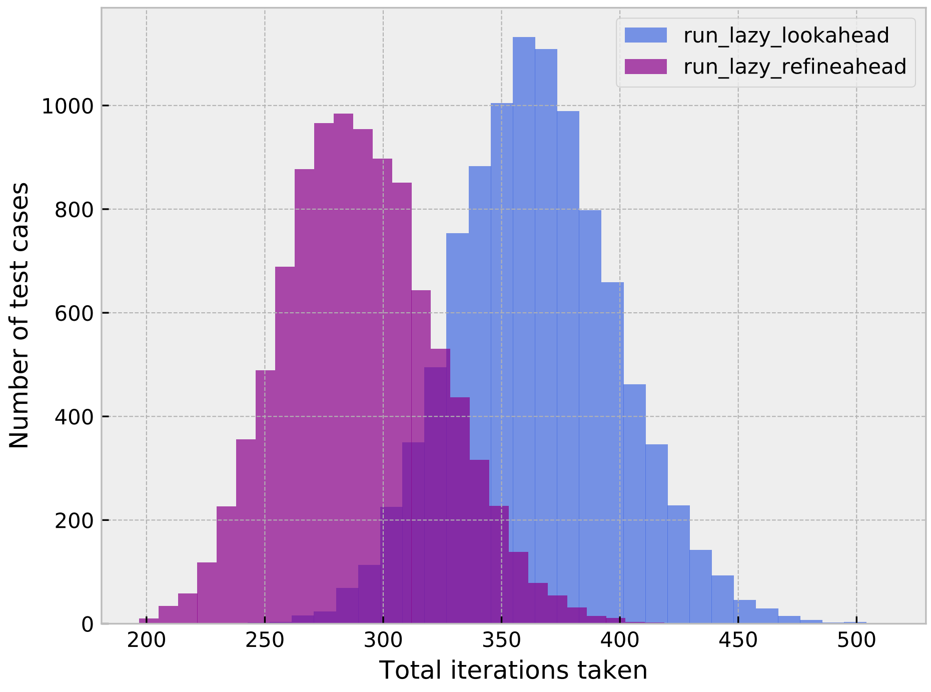


(a)

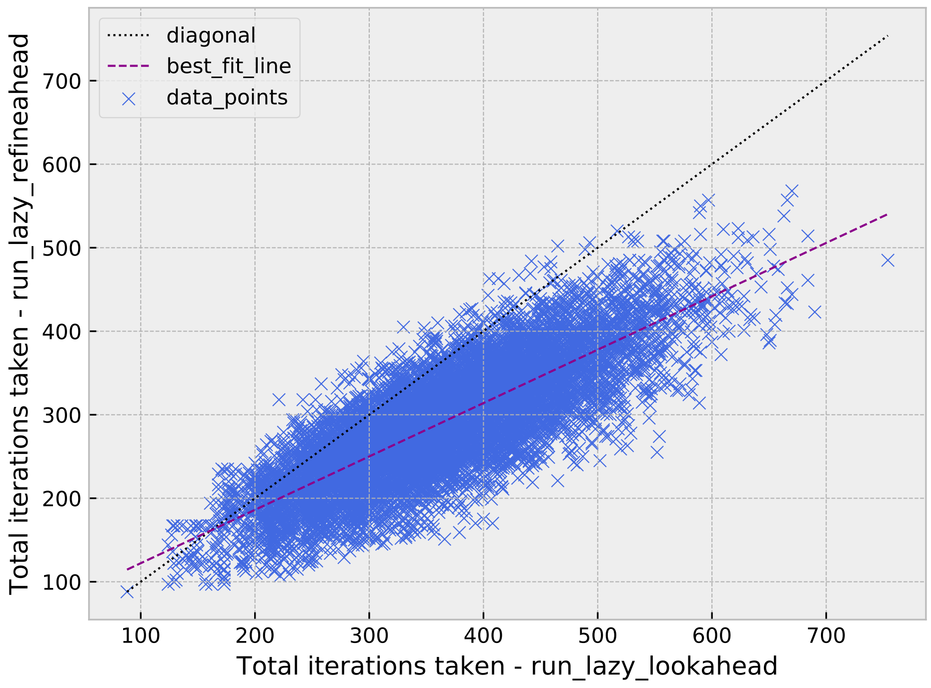


(b)

Figure 4.3: Results of metric - Total actions planned (a) Distribution visualized using histograms (b) Relation visualized by fitting a line on the scatter plot.

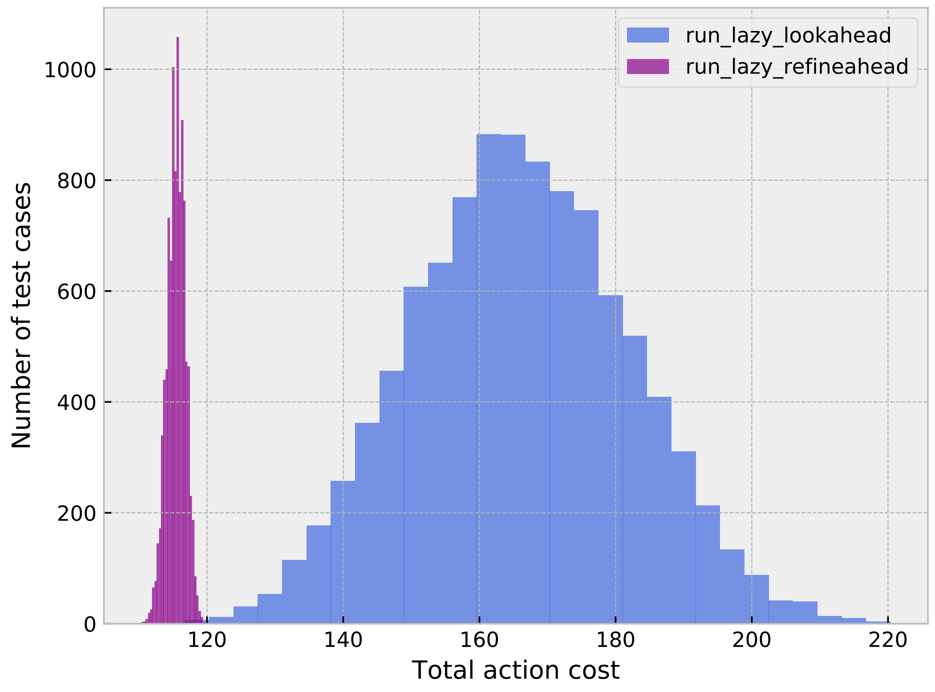


(a)

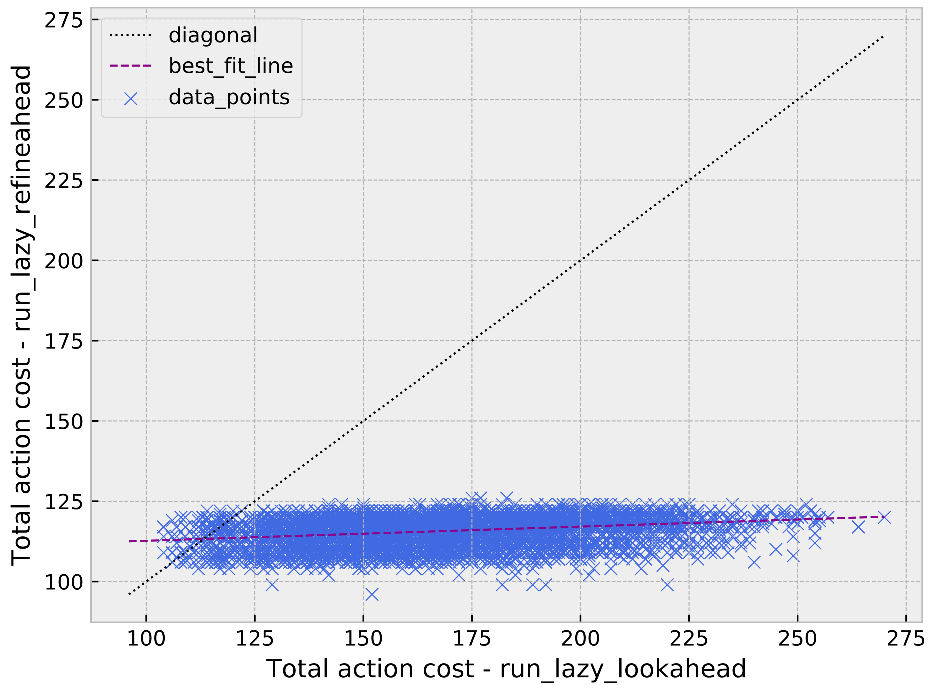


(b)

Figure 4.4: Results of metric - Total iterations taken (a) Distribution visualized using histograms (b) Relation visualized by fitting a line on the scatter plot.

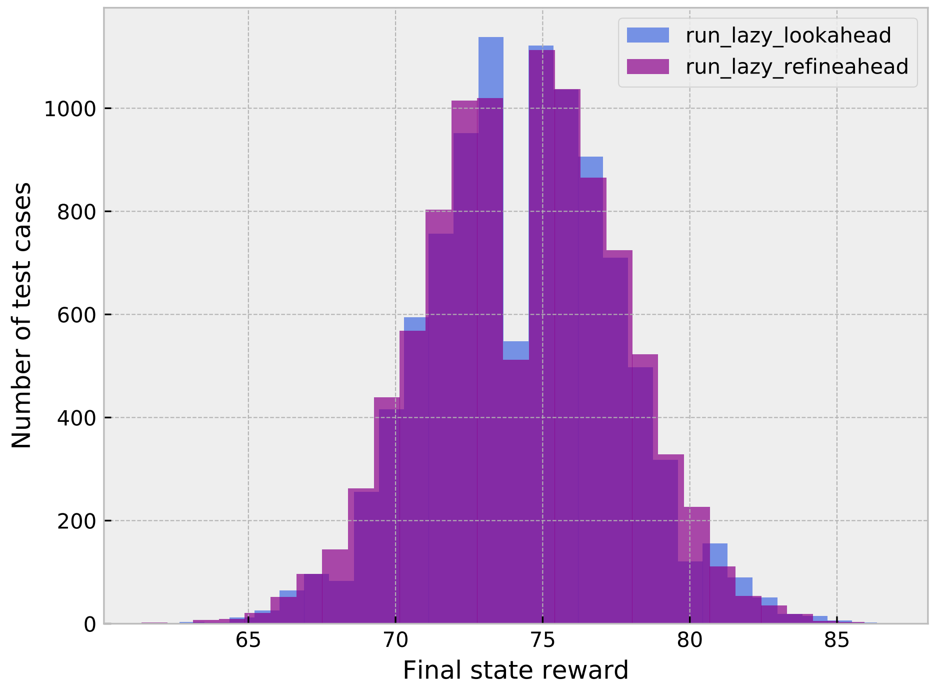


(a)

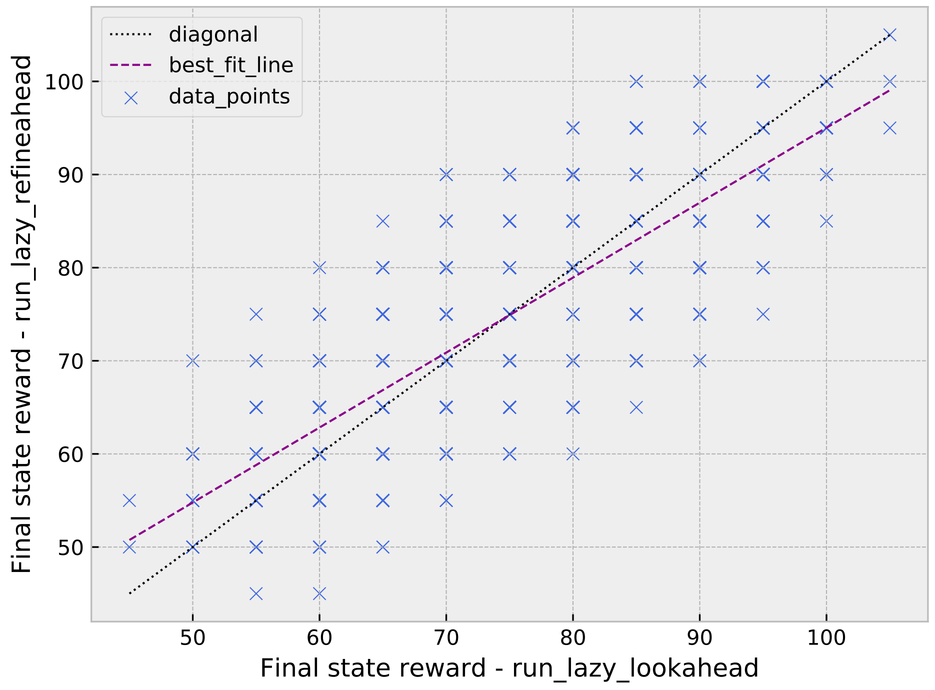


(b)

Figure 4.5: Results of metric - Total action cost (a) Distribution visualized using histograms (b) Relation visualized by fitting a line on the scatter plot.



(a)



(b)

Figure 4.6: Results of metric - Final state reward (a) Distribution visualized using histograms (b) Relation visualized by fitting a line on the scatter plot.

Metric	**Run-Lazy-Lookahead**		**Run-Lazy-Refineahead**	
	Mean	**SD**	**Mean**	**SD**
Total nodes expanded	241.230	21.744	202.006	22.480
Total actions planned	125.509	11.715	115.546	12.716
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 4.1: Overview of results obtained using D_{mean_exp}

Metric	**Refineahead / Lookahead Mean**	**Best-fit line**	
		Slope	**Y-intercept**
Total nodes expanded	0.827	0.685	36.813
Total actions planned	0.893	0.723	24.681
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 4.2: Overview of results obtained using $D_{equivalent}$

and 4.2. The columns in table 4.1 provide the numerical values for the mean and standard deviation of corresponding metrics' histograms. The columns in table 4.2 provide the numerical value of the ratio of metrics' mean for the two algorithms and provide the best-fit line parameters.

4.4 Summary

Based on the results presented in 4.3, we can say that the Run-Lazy-Refineahead is a better algorithm for deliberative HTN acting compared to Run-Lazy-Lookahead. Based on the values of metrics: Total nodes expanded, Total actions planned, and Total iterations taken, we can state that the Run-Lazy-Refineahead leads to the generation of shorter and easily solvable re-planning problems. Also, the average time spent in planning during Run-Lazy-Refineahead is $\approx 80\%$ of the average time spent in planning during Run-Lazy-Lookahead. Based on the values of metrics: Total action cost, we can state that the Run-Lazy-Refineahead leads to the execution of smaller and cheaper action sequences. 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. All these improvements were realized without sacrificing the average final state reward obtained based on the task execution.

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 you to account 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, even after a significant effort is put, 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 was designed primarily for use in the Run-Lazy-Lookahead algorithm. Then the same domain model and state definition were used for the Run-Lazy-Refineahead algorithm. This reuse of domain 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 were 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 models are being used for the task refinement process.

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

Chapter 5: Conclusion

In this thesis, we presented a novel set of algorithms for HTN planning, acting, and integrated planning and acting.

The first main contribution of this thesis 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 models 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 of this thesis is an HTN actor, RAE-lite. RAE-lite is an iterative tree traversal-based HTN acting algorithm written in Python.

RAE-lite is purely reactive and is inspired by IPyHOP's HTN refinement and RAE's task execution. It provides native support to HTN acting for domain models written for IPyHOP and is an excellent alternative for problems where deliberation is not required. At present, very few implementations of hierarchical actors are available in Python. We expect RAE-lite to be useful to the vast robotics community developing their robotic systems using Python. RAE-lite, along with IPyHOP, is also a natural choice for systems that require a mixture of HTN planning and HTN acting.

Finally, the third main contribution of this thisis 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 proved it to be a better alternative to Run-Lazy-Lookahead, another popular repeated planning and acting algorithm, 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 in the overall execution performance 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, RAE-lite, and Run-Lazy-Refineahead for HTN planning, acting, and integrated planning and acting.

5.1 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.

Some of the well-recognized inadequacies of HTN planning are:

- Enormous domain engineering effort in writing HTN methods: This is because the HTN formalism requires users to implement methods to cover every possible scenario that the agent could encounter. If the HTN planner finds itself in a state the user had not anticipated, it will misbehave or fail without returning a solution.

- Brittleness in open and dynamic environments: The previous problem is intensified in open, dynamic environments. Events outside of the agent's control can happen non-deterministically, leading to novel situations not anticipated by the user. HTN planners are not well suited to work in open and dynamic environments.

- Difficulty in designing domain-independent HTN planning heuristics: Heuristics are crucial in guiding the algorithm quickly towards high-quality solutions. Due to the lack of such heuristics, HTN planners are often entirely reliant on the user-provided knowledge through the definition of methods in providing the necessary guidance. Thus further increasing the burden on the user.

IPyHOP, being an HTN planner, also faces all these problems, and not much work has been done to address them. We believe that they are the main areas for future research in improving IPyHOP.

Hierarchical actors like RAE use operational representations, which can be very useful in modeling the acting behavior of an agent. However, RAE-lite uses limited operational representations, which forces it to perform task refinement similar to an HTN planner. On the one hand, RAE-lite's use of extended HTN methods allows seamless cross-compatibility and reuse of the defined domain model for HTN planning and HTN acting; on the other hand, this means that RAE-lite's methods do not possess the expressiveness of operational models. Also, RAE-lite does not allow parallel refinement and task execution the way RAE does. Moreover, unlike RAE, external event handling is done sequentially and synchronously. Our work on RAE-lite was started with the belief that we could design a hierarchical actor that provides a subset of functionalities that RAE does while reducing the implementational complexity disproportionately. The subset of functionalities that we chose to include for RAE-lite was primarily based on our needs. Thus we believe other RAE-like algorithms can be devised that trade-off a different set of functionalities to reduce the implementational complexity.

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. There are algorithms like SeRPE that directly integrate a planner's descriptive model into a hierarchical actor to select refinement methods. Whereas others like [10,11] directly integrate planners that plan using operational representations with the actor RAE. 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. Although not explained in this thesis, both IPyHOP and RAE-lite have been extended to support hierarchical goal planning using Hierarchical Goal Networks (HGN), taking inspiration from GNPyhop. However, all our assertions on the benefits of Run-Lazy-Refineahead over Run-Lazy-Lookahead were with the assumption that we were working in an HTN domain and not an HGN domain. A comparison between these two algorithms for an HGN domain remains to be addressed.

Appendix A: RoboSub Domain in IPyHOP

A.1 Method Definitions

```python
#!/usr/bin/env python
"""
File Description: Robosub methods file. All the methods for Robosub planning
↪  domain are defined here.
"""
# *********************    Libraries to be imported    ******************** #
from ipyhop import Methods

# *********************        Method Definitions      ******************** #
methods = Methods()


def tm1_move(state, loc_):
    if state.loc['r'] != loc_:
        if state.found[loc_] is True:
            return [('a_move', loc_)]
        else:
            if loc_ in state.rigid['adj'][state.loc['r']]:
                return [('a_search_for', loc_), ('a_move', loc_)]
            if loc_ > state.loc['r']:
                l_ = state.rigid['adj'][state.loc['r']][-1]
                return [('a_search_for', l_), ('a_move', l_), ('move_task',
                ↪  loc_)]
            if loc_ < state.loc['r']:
                l_ = state.rigid['adj'][state.loc['r']][0]
                return [('a_search_for', l_), ('a_move', l_), ('move_task',
                ↪  loc_)]
    return []


methods.declare_task_methods('move_task', [tm1_move])


def tm1_cross_gate(state, gate_):
    if state.crossed_gate[gate_] is False:
        if state.loc['r'] == state.loc[gate_]:
            if state.found[gate_] is True:
                return [('a_cross_gate_40', gate_)]
            return [('a_localize', gate_), ('a_cross_gate_40', gate_)]
        return [('move_task', state.loc[gate_]), ('cross_gate_task', gate_)]
    return []


def tm2_cross_gate(state, gate_):
    if state.crossed_gate[gate_] is False:
        if state.loc['r'] == state.loc[gate_]:
            if state.found[gate_] is True:
                return [('a_cross_gate_60', gate_)]
            return [('a_localize', gate_), ('a_cross_gate_60', gate_)]
        return [('move_task', state.loc[gate_]), ('cross_gate_task', gate_)]
    return []


methods.declare_task_methods('cross_gate_task', [tm1_cross_gate, tm2_cross_gate])


def tm1_pick(state, obj_):
    if state.loc[obj_] != 'r':
        if state.loc['r'] == state.loc[obj_]:
            if state.found[obj_]:
                return [('a_pick', obj_)]
            return [('a_localize', obj_), ('a_pick', obj_)]
        return [('move_task', state.loc[obj_]), ('pick_task', obj_)]
    return []


def tm2_pick(*_):
    return []   # skip the task


methods.declare_task_methods('pick_task', [tm1_pick, tm2_pick])


def tm1_trace_path(state, gp_):
    if state.traversed_path[gp_] is False:
        if state.loc['r'] == state.loc[gp_]:
            if state.found[gp_] is True:
                return [('a_trace_guide_path', gp_)]
            return [('a_localize', gp_), ('a_trace_guide_path', gp_)]
        return [('move_task', state.loc[gp_]), ('trace_path_task', gp_)]
    return []


def tm2_trace_path(*_):
    return []   # skip the task


methods.declare_task_methods('trace_path_task', [tm1_trace_path, tm2_trace_path])


def tm1_slay_vampire(state, v_):
    if state.vampire_touched[v_] is False:
        if state.loc['r'] == state.loc[v_]:
            if state.found[v_] is True:
                return [('a_touch_back_v', v_)]
            return [('a_localize', v_), ('a_touch_back_v', v_)]
        return [('move_task', state.loc[v_]), ('slay_vampire_task', v_)]
    return []


def tm2_slay_vampire(state, v_):
    if state.vampire_touched[v_] is False:
        if state.loc['r'] == state.loc[v_]:
            if state.found[v_] is True:
                return [('a_touch_front_v', v_)]
            return [('a_localize', v_), ('a_touch_front_v', v_)]
        return [('move_task', state.loc[v_]), ('slay_vampire_task', v_)]
    return []


def tm3_slay_vampire(*_):
    return []   # skip the task


methods.declare_task_methods('slay_vampire_task', [tm1_slay_vampire,
↪   tm2_slay_vampire, tm3_slay_vampire])


def tm1_drop_garlic(state, gm_, c_):
    if len(state.coffin_filled[c_]) < 2:
        if state.loc[gm_] == 'r':
            if state.loc['r'] == state.loc[c_]:
                if state.found[c_] is True:
                    if state.opened[c_] is True:
                        return [('a_drop_garlic_open_coffin', gm_, c_)]
                    return [('a_open_c', c_), ('a_drop_garlic_open_coffin', gm_,
                    ↪   c_)]
                return [('a_localize', c_), ('a_open_c', c_),
                ↪   ('a_drop_garlic_open_coffin', gm_, c_)]
            return [('move_task', state.loc[c_]), ('drop_garlic_task', gm_, c_)]
        return
    return []


def tm2_drop_garlic(state, gm_, c_):
    if len(state.coffin_filled[c_]) < 2:
        if state.loc[gm_] == 'r':
            if state.loc['r'] == state.loc[c_]:
                if state.found[c_] is True:
                    return [('a_drop_garlic_closed_coffin', gm_, c_)]
                return [('a_localize', c_), ('a_drop_garlic_closed_coffin', gm_,
                ↪   c_)]
            return [('move_task', state.loc[c_]), ('drop_garlic_task', gm_, c_)]
        return
    return []
```

```python

def tm3_drop_garlic(*_):
    return []   # skip the task


methods.declare_task_methods('drop_garlic_task', [tm1_drop_garlic,
↪  tm2_drop_garlic, tm3_drop_garlic])


def tm1_stake_heart(state, t, d):
    if len(state.staked_dracula[d]) < 2 and state.loc[t] == 'r':
        if state.loc['r'] == state.loc[d]:
            if state.found[d] is True:
                if state.decapitated[d] is True:
                    return [('a_stake_decap_d', t, d)]
                return [('a_decap_d', d), ('a_stake_decap_d', t, d)]
            return [('a_localize', d), ('a_decap_d', d), ('a_stake_decap_d', t,
            ↪  d)]
        return [('move_task', state.loc[d]), ('stake_heart_task', t, d)]
    return []


def tm2_stake_heart(state, t, d):
    if len(state.staked_dracula[d]) < 2 and state.loc[t] == 'r':
        if state.loc['r'] == state.loc[d]:
            if state.found[d] is True:
                return [('a_stake_norm_d', t, d)]
            return [('a_localize', d), ('a_stake_norm_d', t, d)]
        return [('move_task', state.loc[d]), ('stake_heart_task', t, d)]
    return []


def tm3_stake_heart(*_):
    return []   # skip the task


methods.declare_task_methods('stake_heart_task', [tm1_stake_heart,
↪  tm2_stake_heart, tm3_stake_heart])


def tm1_surface(state, s_):
    if state.surfaced['r'] is False:
        if state.loc['r'] == state.loc[s_]:
            if state.found[s_] is True:
                return [('a_surface', 'cm1', s_)]
            return [('a_localize', s_), ('a_surface', 'cm1', s_)]
        return [('move_task', state.loc[s_]), ('surface_task', s_)]
    return []


def tm2_surface(state, s_):
    if state.surfaced['r'] is False:
        if state.loc['r'] == state.loc[s_]:
            if state.found[s_] is True:
                return [('a_surface', 'cm2', s_)]
            return [('a_localize', s_), ('a_surface', 'cm2', s_)]
        return [('move_task', state.loc[s_]), ('surface_task', s_)]
    return []


def tm3_surface(*_):
    return []   # skip the task


methods.declare_task_methods('surface_task', [tm1_surface, tm2_surface,
↪  tm3_surface])


def tm1_pinger(state):
    tasks = []
    if state.found['ap1'] is False:
        tasks.append(('a_localize_ap', 'ap1'))
    if state.found['ap2'] is False:
        tasks.append(('a_localize_ap', 'ap2'))
    return tasks


methods.declare_task_methods('pinger_task', [tm1_pinger])


def _common_tasks(state, task_list, locs):
    if state.loc['g'] == locs and state.crossed_gate['g'] is False:
        task_list.append(('cross_gate_task', 'g'))

    if state.loc['gm1'] == locs:
        task_list.append(('pick_task', 'gm1'))
    if state.loc['gm2'] == locs:
        task_list.append(('pick_task', 'gm2'))
    if state.loc['c1'] == locs:
        if state.loc['gm1'] != 'c1':
            task_list.append(('drop_garlic_task', 'gm1', 'c1'))
        if state.loc['gm2'] != 'c1':
            task_list.append(('drop_garlic_task', 'gm2', 'c1'))

    if state.loc['cm1'] == locs:
        task_list.append(('pick_task', 'cm1'))
    if state.loc['cm2'] == locs:
        task_list.append(('pick_task', 'cm2'))

    if state.loc['v1'] == locs and state.vampire_touched['v1'] is False:
        task_list.append(('slay_vampire_task', 'v1'))
    if state.loc['v2'] == locs and state.vampire_touched['v2'] is False:
        task_list.append(('slay_vampire_task', 'v2'))

    if state.loc['d1'] == locs:
        if state.loc['t1'] == 'r':
            task_list.append(('stake_heart_task', 't1', 'd1'))
        if state.loc['t2'] == 'r':
```

```python
            task_list.append(('stake_heart_task', 't2', 'd1'))

    if state.loc['gp1'] == locs and state.traversed_path['gp1'] is False:
        task_list.append(('trace_path_task', 'gp1'))
    if state.loc['gp2'] == locs and state.traversed_path['gp2'] is False:
        task_list.append(('trace_path_task', 'gp2'))

    if state.loc['s1'] == locs and state.surfaced['r'] is False:
        task_list.append(('surface_task', 's1'))


def tm1_main(state, loc_list):
    task_list = []
    for locs in loc_list:
        task_list.append(('move_task', locs))
        _common_tasks(state, task_list, locs)

    return task_list


methods.declare_task_methods('main_task', [tm1_main])

# *********************    Demo / Test Routine         ********************* #
if __name__ == '__main__':
    raise NotImplementedError("Test run / Demo routine for Robosub Mod Methods
    ↪  isn't implemented.")

"""
Author(s): Yash Bansod
Repository: https://github.com/YashBansod/IPyHOP
Organization: University of Maryland at College Park
"""
```

A.2 Action Definitions

```python
#!/usr/bin/env python
"""
File Description: Robosub actions file. All the actions for Robosub planning
↪  domain are defined here.
"""
# *********************    Libraries to be imported    ********************* #
from ipyhop import Actions


# *********************        Action Definitions      ********************* #
actions = Actions()


# search for a location in the field
def a_search_for(state, loc_):
    if loc_ in state.rigid['adj'][state.loc['r']]:
        state.found[loc_] = True
        return state


# search for an object at current location
def a_localize(state, obj_):
    if state.loc['r'] == state.loc[obj_]:
        state.found[obj_] = True
        return state


# acoustic pinger triangulation
def a_localize_ap(state, ap_):
    if state.rigid['type'][ap_] == 'ap':
        state.found[ap_] = True
        state.found[state.loc[ap_]] = True
        return state


# move to a recognized location
def a_move(state, loc_):
    if state.found[loc_] is True and state.rigid['type'][loc_] == 'l':
        state.loc['r'] = loc_
        return state


# cross the gate at current location from 40% side
def a_cross_gate_40(state, gate_):
    if state.loc['r'] == state.loc[gate_] and state.found[gate_] is True and
    ↪  state.rigid['type'][gate_] == 'g':
        state.crossed_gate[gate_] = 'T40'
        return state


# cross the gate at current location from 60% side
def a_cross_gate_60(state, gate_):
    if state.loc['r'] == state.loc[gate_] and state.found[gate_] is True and
    ↪  state.rigid['type'][gate_] == 'g':
        state.crossed_gate[gate_] = 'T60'
        return state


# pick an obj at current location (allowed objects: crucifix marker, garlic
↪  marker)
def a_pick(state, obj_):
    if state.loc['r'] == state.loc[obj_] and state.found[obj_] is True:
        if state.rigid['type'][obj_] == 'gm' or state.rigid['type'][obj_] ==
        ↪  'cm':
            state.loc[obj_] = 'r'
            return state


# trace a guide path at current location
def a_trace_guide_path(state, gp_):
    if state.loc['r'] == state.loc[gp_] and state.found[gp_] is True and state.rigid['type'][gp_] == 'gp':
        state.traversed_path[gp_] = True
        return state


# touch the back of a vampire at current location
def a_touch_back_v(state, v_):
    if state.loc['r'] == state.loc[v_] and state.found[v_] is True and state.rigid['type'][v_] == 'v':
        state.vampire_touched[v_] = 'Tb'
        return state


# touch the front of a vampire at current location.
def a_touch_front_v(state, v_):
    if state.loc['r'] == state.loc[v_] and state.found[v_] is True and state.rigid['type'][v_] == 'v':
        state.vampire_touched[v_] = 'Tf'
        return state


# open the coffin at current location
def a_open_c(state, c_):
    if state.loc['r'] == state.loc[c_] and state.found[c_] is True and state.rigid['type'][c_] == 'c':
        state.opened[c_] = True
        return state


# drop a garlic in an opened coffin at current location
def a_drop_garlic_open_coffin(state, gm_, c_):
    if state.loc[gm_] == 'r' and state.loc['r'] == state.loc[c_] and state.opened[c_] is True:
        if state.rigid['type'][c_] == 'c' and state.rigid['type'][gm_] == 'gm':
            state.loc[gm_] = c_
            state.coffin_filled[c_].append('1o')
            return state


# drop garlic on a closed coffin at current location
def a_drop_garlic_closed_coffin(state, gm_, c_):
    if state.loc[gm_] == 'r' and state.loc['r'] == state.loc[c_] and state.found[c_] is True:
        if state.rigid['type'][c_] == 'c' and state.rigid['type'][gm_] == 'gm':
            state.loc[gm_] = c_
            state.coffin_filled[c_].append('1c')
            return state


# decapitate a dracula at current location
def a_decap_d(state, d_):
    if state.loc['r'] == state.loc[d_] and state.found[d_] is True and state.rigid['type'][d_] == 'd':
        state.decapitated[d_] = True
        return state


# stake a decapitated dracula at current location
def a_stake_decap_d(state, t_, d_):
    if state.loc[t_] == 'r' and state.loc['r'] == state.loc[d_] and
    ↪  state.decapitated[d_] is True:
        if state.rigid['type'][t_] == 't' and state.rigid['type'][d_] == 'd':
            state.loc[t_] = d_
            state.staked_dracula[d_].append('1d')
            return state


# stake a normal dracula at current location
def a_stake_norm_d(state, t_, d_):
    if state.loc[t_] == 'r' and state.loc['r'] == state.loc[d_] and
    ↪  state.found[d_] is True:
        if state.rigid['type'][t_] == 't' and state.rigid['type'][d_] == 'd':
            state.loc[t_] = d_
            state.staked_dracula[d_].append('1n')
            return state


# surface in a surface zone at current location carrying a crucifix marker
def a_surface(state, cm, s):
    if state.loc['r'] == state.loc[s] and state.loc[cm] == 'r' and state.found[s]
    ↪  is True:
        if state.rigid['type'][cm] == 'cm' and state.rigid['type'][s] == 's':
            state.surfaced['r'] = True
            return state


actions.declare_actions([a_search_for, a_localize, a_localize_ap, a_move,
↪  a_cross_gate_40, a_cross_gate_60, a_pick,
                         a_touch_back_v, a_touch_front_v, a_trace_guide_path,
                         ↪  a_open_c, a_drop_garlic_open_coffin,
                         a_drop_garlic_closed_coffin, a_decap_d, a_stake_decap_d,
                         ↪  a_stake_norm_d, a_surface])

action_probability = {
    'a_cross_gate_40': [0.3, 0.7],
    'a_pick': [0.95, 0.05],
    'a_touch_back_v': [0.4, 0.6],
    'a_touch_front_v': [0.8, 0.2],
    'a_trace_guide_path': [0.85, 0.15],
    'a_open_c': [0.5, 0.5],
    'a_drop_garlic_open_coffin': [0.9, 0.1],
    'a_drop_garlic_closed_coffin': [0.9, 0.1],
    'a_decap_d': [0.4, 0.6],
    'a_stake_decap_d': [0.8, 0.2],
    'a_stake_norm_d': [0.8, 0.2],
}
```

```python
action_cost = {
    'a_search_for': 2,
    'a_move': 5,
    'a_cross_gate_40': 10,
    'a_cross_gate_60': 8,
    'a_pick': 3,
    'a_touch_front_v': 3,
    'a_touch_back_v': 6,
    'a_trace_guide_path': 3,
    'a_open_c': 5,
    'a_drop_garlic_open_coffin': 2,
    'a_drop_garlic_closed_coffin': 2,
    'a_decap_d': 5,
    'a_stake_norm_d': 2,
    'a_stake_decap_d': 2,
    'a_surface': 3,
}

actions.declare_action_models(action_probability, action_cost)

# *********************    Demo / Test Routine         ********************* #
if __name__ == '__main__':
    raise NotImplementedError("Test run / Demo routine for Robosub Mod Commands
    ↪  isn't implemented.")

"""
Author(s): Yash Bansod
Repository: https://github.com/YashBansod/IPyHOP
Organization: University of Maryland at College Park
"""
```

Bibliography

[1] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: theory and practice*. Elsevier, 2004.

[2] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated planning and acting*. Cambridge University Press, 2016.

[3] Dana Nau, T-C Au, Okhtay Ilghami, Ugur Kuter, Dan Wu, Fusun Yaman, Héctor Munoz-Avila, and J William Murdock. Applications of SHOP and SHOP2. *IEEE Intelligent Systems*, 20(2):34–41, 2005.

[4] Ilche Georgievski and Marco Aiello. HTN planning: Overview, comparison, and beyond. *Artificial Intelligence*, 222:124–156, 2015.

[5] Fletcher Thompson and Damien Guihen. Review of mission planning for autonomous marine vehicle fleets. *Journal of Field Robotics*, 36(2):333–354, 2019.

[6] Derek Long and Maria Fox. The 3rd international planning competition: Results and analysis. *Journal of Artificial Intelligence Research*, 20:1–59, 2003.

[7] John Paul Kelly, Adi Botea, Sven Koenig, et al. Offline planning with hierarchical task networks in video games. In *AIIDE*, pages 60–65, 2008.

[8] 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.

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

[10] Sunandita Patra, Malik Ghallab, Dana Nau, and Paolo Traverso. Acting and planning using operational models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 7691–7698, 2019.

[11] Sunandita Patra, James Mason, Amit Kumar, Malik Ghallab, Paolo Traverso, and Dana Nau. Integrating acting, planning, and learning in hierarchical operational models. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30, pages 478–487, 2020.

[12] Dana Nau, Yue Cao, Amnon Lotem, and Hector Munoz-Avila. SHOP: Simple hierarchical ordered planner. In *Proceedings of the 16th international joint conference on Artificial intelligence-Volume 2*, pages 968–973, 1999.

[13] Dana S Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J William Murdock, Dan Wu, and Fusun Yaman. SHOP2: An HTN planning system. *Journal of artificial intelligence research*, 20:379–404, 2003.

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

[15] Dana Nau. Game applications of HTN planning with state variables. In *Planning in Games: Papers from the ICAPS Workshop*, 2013.

[16] Jeff Orkin. Three states and a plan: the AI of FEAR. In *Game developers conference*, volume 2006, page 4, 2006.

[17] Richard E Fikes and Nils J Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

[18] 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.

[19] Earl D Sacerdoti. The nonlinear nature of plans. Technical report, Stanford Research Inst Menlo Park CA, 1975.

[20] Austin Tate. Generating project networks. In *Proceedings of the 5th international joint conference on Artificial intelligence-Volume 2*, pages 888–893, 1977.

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

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

[23] Austin Tate, Brian Drabble, and Richard Kirby. O-Plan2: an open architecture for command, planning and control. In *Intelligent Scheduling*. Citeseer, 1994.

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

[25] Luis Castillo, Juan Fdez-Olivares, Óscar García-Pérez, and Francisco Palao. Temporal enhancements of an HTN planner. In *Conference of the Spanish Association for Artificial Intelligence*, pages 429–438. Springer, 2005.

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

[27] Zohar Feldman and Carmel Domshlak. Monte-Carlo planning: Theoretically fast convergence meets practical efficiency. *arXiv preprint arXiv:1309.6828*, 2013.

[28] Zohar Feldman and Carmel Domshlak. Monte-Carlo tree search: To MC or to DP? In *ECAI*, pages 321–326, 2014.

[29] Sung Wook Yoon, Alan Fern, and Robert Givan. FF-Replan: A baseline for probabilistic planning. In *ICAPS*, volume 7, pages 352–359, 2007.

[30] Florent Teichteil-Koenigsbuch, Guillaume Infantes, and Ugur Kuter. RFF: A robust, FF-based MDP planning algorithm for generating policies with low probability of failure. *Sixth International Planning Competition at ICAPS*, 8, 2008.

[31] Sung Wook Yoon, Alan Fern, Robert Givan, and Subbarao Kambhampati. Probabilistic planning via determinization in hindsight. In *AAAI*, pages 1010–1016, 2008.

[32] François Félix Ingrand, Raja Chatila, Rachid Alami, and Frédéric Robert. PRS: A high level supervision and control language for autonomous mobile robots. In *Proceedings of IEEE International Conference on Robotics and Automation*, volume 1, pages 43–49. IEEE, 1996.

[33] Olivier Despouys and François Félix Ingrand. Propice-plan: Toward a unified framework for planning and execution. In *European Conference on Planning*, pages 278–293. Springer, 1999.

[34] R James Firby. An investigation into reactive planning in complex domains. In *AAAI*, volume 87, pages 202–206, 1987.

[35] Reid G Simmons. Concurrent planning and execution for autonomous robots. *IEEE Control Systems Magazine*, 12(1):46–50, 1992.

[36] Reid Simmons and David Apfelbaum. A task description language for robot control. In *Proceedings. 1998 IEEE/RSJ International Conference on Intelligent Robots and Systems. Innovations in Theory, Practice and Applications (Cat. No. 98CH36190)*, volume 3, pages 1931–1937. IEEE, 1998.

[37] Michael Beetz and Drew V McDermott. Improving robot plans during their execution. In *AIPS*, pages 7–12, 1994.

[38] Nicola Muscettola, P Pandurang Nayak, Barney Pell, and Brian C Williams. Remote agent: To boldly go where no AI system has gone before. *Artificial intelligence*, 103(1-2):5–47, 1998.

[39] Karen L Myers. CPEF: A continuous planning and execution framework. *AI Magazine*, 20(4):63–63, 1999.

[40] Vandi Verma, Tara Estlin, Ari Jónsson, Corina Pasareanu, Reid Simmons, and Kam Tso. Plan execution interchange language (PLEXIL) for executable plans and command sequences. In *International symposium on artificial intelligence, robotics and automation in space (iSAIRAS)*, 2005.

[41] Fei-Yue Wang, Konstantinos J Kyriakopoulos, Athanasios Tsolkas, and George N Saridis. A petri-net coordination model for an intelligent mobile robot. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(4):777–789, 1991.

[42] Jonathan Bohren, Radu Bogdan Rusu, E Gil Jones, Eitan Marder-Eppstein, Caroline Pantofaru, Melonee Wise, Lorenz Mösenlechner, Wim Meeussen, and Stefan Holzer. Towards autonomous robotic butlers: Lessons learned with the PR2. In *2011 IEEE International Conference on Robotics and Automation*, pages 5568–5575. IEEE, 2011.

[43] David J Musliner, Michael JS Pelican, Robert P Goldman, Kurt D Krebsbach, and Edmund H Durfee. The evolution of CIRCA, a theory-based AI architecture with real-time performance guarantees. In *AAAI Spring Symposium: Emotion, Personality, and Social Behavior*, volume 1205, 2008.

[44] Raphaël Lallement, Lavindra De Silva, and Rachid Alami. HATP: An HTN planner for robotics. *arXiv preprint arXiv:1405.5345*, 2014.

[45] Jason Wolfe, Bhaskara Marthi, and Stuart Russell. Combined task and motion planning for mobile manipulation. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 20, 2010.