Structured Representations for Behaviors of Autonomous Robots

Aaron M. Roth

CMU-RI-TR-19-50

July 15, 2019

The Robotics Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA. 15213

Thesis Committee:
Herbert A. Simon University Professor Manuela M. Veloso, *chair*
Associate Research Professor Aaron Steinfeld
Nicholay Topin

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Robotics.

Abstract

As an increasing number of robot platforms and robot tasks become available and feasible, we aim to improve methods for modular robot task definition and execution. Regardless of the source (human or learned), autonomous robot behavior can be captured in many ways: as code, as modules of code, in an unstructured form such as a neural net, or in one of several more structured formats such as a graph, table, or tree.

This thesis explores structured representations that are simultaneously understandable by humans and executable by robots. Enforcing a certain *structure* on policies can streamline the development of code, enable task transfer, facilitate task instruction through other modalities such as interactive dialogue, and cause autonomously learned policies to be interpretable. This effort is further motivated by the growing desire in the field of reinforcement learning (and machine learning in general) to move from black-box models toward "interpretable AI."

We present Transferable Augmented Instruction Graphs (TAIGs), a platform-independent task representation and execution framework based on the functional composition of robot behavioral and perceptual primitives. We provide an overview of the previously introduced Instruction Graphs and contribute the Augmented Instruction Graphs with the ability to use memory and represent negated conditions, halt conditions, and nested graphs in order to capture complex task policies. We further define representation and execution management to reference a library of primitives allowing policies to be transferred between different robot platforms. We demonstrate the use of TAIGs by applying them to a concrete matching game task example using two autonomous robots: Pepper and Baxter. We further demonstrate TAIG in the context of performing the RoboCup@Home General Purpose Service Robot challenge task.

Recognizing the value of having a means of constructing a graph aside from programming, we introduce Interactive-TAIG, a framework for enabling construction of TAIGs through an interactive dialogue. We demonstrate this construction technique with a simple search-and-deliver task.

We discuss two types of structured representations for policies learned autonomously via reinforcement learning. The first is a decision tree structure, where we extend the partial-Conservative Q-Improvement (pCQI) method into two successive methods: Conservative Q-Improvement and

Conservative Q-Improvement 2. The class of decision tree policies learned via reinforcement learning that we discuss in this thesis are the class of policies that consists of a decision tree over the state space, which requires fewer parameters to express than traditional policy representations. In contrast to many existing methods for creating decision tree policies via reinforcement learning, which focus on accurately representing an action-value function during training, our extension of the pCQI algorithm only increases tree size when the estimated discounted future reward of the overall policy would increase by a sufficient amount. Through evaluation in simulated environment, we show that its performance is comparable or superior to non-CQI-based methods. Additionally, we discuss tuning parameters to control the tradeoff between optimizing for smaller tree size or for overall reward.

Secondly, we introduce a method for learning a TAIG using reinforcement learning. The resulting TAIG includes WHILE loops in the structure, corresponding to subtasks of the task. This method is a means of a robot autonomously learning a policy that then has all the benefits of a TAIG.

This thesis includes the release of a few open-source projects and pieces of code. We release an open-source Python library implementing the TAIG and interactive-TAIG contributions. Also included are tutorials and examples for using TAIG on an arbitrary robotic system. Finally, we release an open-source library with a new AI gym-compatible environment where the agent controls traffic lights in four-way intersection.

Acknowledgments

My adviser, Manuela Veloso, has not only guided me in my research but has helped me to grow as a scientist and researcher. I am grateful for the opportunities she gave me, and I hope to take forward what I have learned with her to the next step in my journey.

Aaron Steinfeld was very helpful in my research even before he joined my thesis committee. His door was open, and he would share his insights and experience with me as we discussed my work.

I extend warm gratitude to the entire CORAL group. In particular, thank you to Michiel de Jong and Kevin Zhang for acquainting me with the Pepper robot. I will always remember the technical battles we later fought alongside each other. A big thank you to Nicholay Topin, who has served as a collaborator and also as a member of my thesis committee. Anahita Kabir was often willing to lend me her ear when I had questions, whether technical or meta-academic, and she helped take a video for the interactive TAIG demonstration in this thesis. I also appreciate my conversations with Ashwin Khadke, Philip Cooksey, Vittorio Perera, Devin Schwab, and Rui Silva. Thank you to Robin Schmucker and João Cartucho for creating the card matching game used in one of the TAIG demonstrations in this thesis.

Many thanks to Fei Fang and Afsaneh Doryab for supporting the Pepper Affective Language project. Thank you my fellow students on that large project, especially Samantha Reig.

I appreciated the time given to me and thoughts shared by Maxim Likhachev, David Garlan, and Pooyan Jamshidi (of the University of South Carolina), with whom I had the opportunity to work on various smaller projects during my time at CMU.

Good research cannot happen without the proper infrastructure. I thank BJ Fecich for helping the Robotics Institute to run smoothly and Sharon Cavlovich for supporting the CORAL group.

Thank you to my parents, who were supportive of my crazy decision to leave industry, return to academia, and begin this journey of learning.

Funding

Some of the material included in this thesis is based upon work supported by Air Force Research Laboratory contract FA87501620042 and ONR grant N000141712899. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of AFRL or ONR.

Contents

1 Introduction **1**
- 1.1 Motivation and Approach 1
- 1.2 Guide to Thesis 3

2 Background and Related Work **5**
- 2.1 Development Libraries and Task Representations for Robots 5
- 2.2 Enabling End-Users to Teach Robots Skills by Instruction 6
- 2.3 Reinforcement Learning and Decision Trees 7
  - 2.3.1 Reinforcement Learning 7
  - 2.3.2 Decision Trees 7
  - 2.3.3 Learning Decision Tree Policies with RL 8
- 2.4 Summary 9

3 Transferable Augmented Instruction Graph: Formalization **11**
- 3.1 Background: Instruction Graph 12
- 3.2 Augmented Instruction Graph 13
  - 3.2.1 Memory 13
  - 3.2.2 Negation 15
- 3.3 Transferable Augmented Instruction Graph 15
  - 3.3.1 Primitives and Primitive Library 15
  - 3.3.2 Bringing it All Together: The Library-Memory-Graph Paradigm 17
- 3.4 Further Augmentations 19
  - 3.4.1 Nested Graphs (Graphs as Primitives) 19
  - 3.4.2 Halt Conditions 20
- 3.5 Open Source TAIG Library for Use by the General Robotics Community 20
- 3.6 Summary 21

4 Transferable Augmented Instruction Graph: Demonstration **23**
- 4.1 Demonstration and Qualitative Comparison: Task Transfer 23
  - 4.1.1 Robot System 23
  - 4.1.2 Task Demonstration: Pepper Matching Game 24
  - 4.1.3 Robot-Independent Task Transference: Baxter Matching Game 26
- 4.2 Practical Demonstration: General Purpose Service Robot 28
  - 4.2.1 RoboCup@Home and the General Purpose Service Robot Task 28

4.2.2 Robot System 28
4.2.3 Enabling GPSR through TAIG 31
4.2.4 Description of a Specific GSPR Command Execution: "Deliver Object From Location" Task 33
4.3 Further Use of TAIG 38
4.4 Summary 40

5 Interactive Transferable Augmented Instruction Graph 41
5.1 Methods for Realizing Runtime Interactivity 42
5.1.1 The Updated Primitive Tuple 42
5.1.2 The Interactive Manager 45
5.2 Learning to Search and Deliver a Message 49
5.3 Summary 54

6 Explainable Reinforcement Learning via Decision Tree Policies: Using Human-Defined Conditions 55
6.1 Taxi Environment 56
6.2 Transformation of an Existing Tabular Policy 56
6.2.1 Method 56
6.2.2 Exploratory Results 59
6.3 Learning a Tree with Pre-Specified Conditions 61
6.3.1 Method 61
6.3.2 Results 61
6.4 Summary 63

7 Explainable Reinforcement Learning via Decision Tree Policies: Conservative Q-Improvement 65
7.1 Modified RobotNav Environment 66
7.2 Vehicle Intersection Environment 67
7.3 Partial-CQI 69
7.4 CQI and CQI2: Method 72
7.4.1 CQI: Multiple Possible Split Locations Per Dimensions 72
7.4.2 CQI2: Feature Combinations 74
7.5 CQI and CQI2: Results 75
7.5.1 Direct Comparison 76
7.5.2 Parameter Sensitivity 80
7.6 Summary 82

8 Learning a TAIG-style Policy with Reinforcement Learning 83
8.1 Overview of Approach 83
8.2 Methodology 84

8.3 Limitations . . . 88
8.4 Results . . . 88
8.5 Summary . . . 93

9 Conclusions 95
9.1 Contributions . . . 95
9.2 Future Work . . . 97

A Developer's Tutorial for Using TAIG With Pepper 99
A.1 Using TAIG for a Task . . . 99
A.1.1 The Memory Object . . . 100
A.1.2 The Primitive Library . . . 100
A.1.3 Controller: Graph Creation and Execution . . . 101
A.1.4 Let's Run It . . . 102
A.2 Conclusion . . . 103

B Documentation for TAIG Open Source Library 105
B.1 Installation . . . 106
B.2 Introduction . . . 106
B.3 QuickStart / Example . . . 107
B.4 Details for developing on your own TAIG . . . 108
B.4.1 Creating Memory . . . 108
B.4.2 Creating Primitives . . . 109
B.4.3 Creating the Graph and Saving to File . . . 111
B.4.4 Running the Graph . . . 111
B.4.5 Additional Techniques . . . 112
B.4.6 Nested Graphs (Graphs as Primitives) . . . 112
B.5 Interactive TAIG . . . 114
B.5.1 Primitive Library for IM . . . 114
B.5.2 Using the Interactive Manager . . . 116
B.5.3 Builder Phrases . . . 118
B.6 Credits . . . 120

C Documentation for Vehicle Intersection Open Source AI Gym Environment 121
C.1 Description . . . 122
C.2 Action Space . . . 123
C.2.1 ToggleRoad() . . . 123
C.2.2 SetByRoad() . . . 123
C.2.3 ToggleLight() . . . 123
C.2.4 SetByLight() . . . 124

C.2.5 SetExplicitly() 124
C.3 Observation Space 124
C.4 Requirements 125
C.5 Setup 125
C.6 Usage 125
C.7 Example 126
C.8 Settings 126

D Gridsearch Bounds Used and Hyperparameters Found When Learning Decision Tree Policies 129

E TAIGs for Additional Subtask Graphs for GPSR 133

F List of All Primitives Used in the GPSR task 157

Bibliography 161

When this dissertation is viewed as a PDF, the page header is a link to this Table of Contents.

List of Figures

3.1 The structure of an example IG. 12
3.2 The three components of the Memory-Library-Graph paradigm interact. 18

4.1 Primitive Library and Task Graph for Matching Game Task 25
4.2 Pepper and Baxter play the matching game. 27
4.3 Example GPSR Category 2 Commands 29
4.4 Primitives used in GPSR parent TAIG. 32
4.5 Primitives used in "Deliver Object From" TAIG. 39

5.1 A finite state machine representing the Interactive Manager's behavior 47
5.2 Description of primitives in Primitive Library used in the Search and Deliver Message task 50
5.3 Description of the task graphs that are created during the demonstrated dialogue 51
5.4 Full dialogue for the demonstration 52
5.5 Pepper learns and executes the "find-and-remind-me-of-a-meeting" task. 53

6.1 Visual representation of the taxi environment. 56
6.2 Example count of how many states indicate a particular best action for a successful policy 59
6.3 List of conditions used for taxi decision tree. 60
6.4 Taxi environment states 283 and 183 60

7.1 An image of Robot Navigation environment 66
7.2 Two examples of rendering of the vehicle intersection environment . . 67
7.3 Example of an update to the policy tree in pCQI 71
7.4 Example of the multiple Split objects per dimension in CQI 72
7.5 Comparison of reward vs size during training for pCQI vs CQI2 on Vehicle Intersection 78
7.6 Comparison of reward vs size during training for various methods on Vehicle Intersection 79
7.7 Analysis of the effect of the n_S CQI/CQI2 hyperparameter 80
7.8 Analysis of the effect of the H_S CQI/CQI2 hyperparameter 81

8.1 Description of structure of TAIG policy for two-stage task after transformation from decision tree 89
8.2 Illustration of the TAIG transformation process for a subsection of a decision tree 91

A.1 An example of the structure of the Task Graph and Primitive Library. 99
A.2 Functions for use in Primitives 101
A.3 ActionPrimitive and ConditionalPrimitive 101
A.4 Pepper using TAIG 102

C.1 VI rendering example 122

List of Tables

6.1 Example of sufficiency concept: states and features 58

7.1 Comparison of RL decision tree policy methods across environments 77

8.1 Best results for TAIG policy per %-successful-transformation group . 90

B.1 Builder Phrases 119

D.1 Gridsearch bounds for pCQI, CQI, and CQI2 130
D.2 Gridsearch bounds for Pyeatt Method 130
D.3 Best hyperparameters found for various RL decision tree policy methods across environments 131

Chapter 1

Introduction

1.1 Motivation and Approach

The decades ahead will see an increased number of robots performing tasks for the benefit of humans. Teaching or programming a new task for a robot often requires complex, tedious implementation. We envision a future in which it is quicker and easier to instruct robots how to execute new tasks. Robot assistants and partners will also be able to learn tasks autonomously, but in a manner that allows for interpreting the learned policy without having to execute it.

Regardless of the source (human or learned by the robot itself), autonomous robot behavior can be captured in many ways: as code, as modules of code, in an unstructured form such as a neural net, or in one of several more structured formats such as a graph, table, or tree.

An important stepping stone towards the future described above is creating explainable structured representations of policies for the execution of tasks. This thesis is interested, in particular, in structured representations that are simultaneously understandable by humans and executable by robots. When a human and a robot have a common structured representation for components of a task, it becomes easier for a human to communicate what needs to happen to the robot (whether by programming or through a more natural interactive process). When disparate robots can use that common representation, it allows task transfer, where a task meant to be performed by one type of robot can also be performed by another. When an autonomously learned policy is structured, it becomes more interpretable.

This thesis extends and develops techniques with the goal of achieving structured, interpretable policies for robots. This thesis considers tasks involving motion and perception.

One way to move towards this goal is with work on Transferable Augmented Instruction Graphs (TAIG). This thesis proposes taking advantage of the robot's knowledge of how to perform other tasks involving similar functionality, or of the fact that alternate robotic systems could perform the same task. Task dissimilarity or differences in specific hardware and software configurations are existing challenges that TAIG is designed to overcome.

TAIG extends and improves upon Instruction Graph (IG) [47], which represents a task policy as a structured graph. TAIG overcomes some of IG's limitations and realizes additional potential. TAIG brings together the concepts of task transfer, increased ease of task policy creation and maintenance, shared memory between primitives, and instruction via natural conversation. TAIG begins to account for the disparities between robot systems that exist in the real world and the differences between human and robot understanding of a task.

Another way to move towards our envisioned future is through focusing on structured policies in the context of autonomous learning. Much work has been done on autonomous robot learning in general. Within the field, there is a debate as to the relative merits of model-free or model-based methods. One of the major drawbacks of model-free policies is that the policies themselves are somewhat opaque—they have to be executed in order to know what they will do. Many powerful machine learning algorithms have been developed that are “black boxes” [48]. They produce models which can be used to make predictions or policies that can be used to perform tasks, but these models are often inscrutable as to their inner logic, or at least require significant analysis to penetrate. Black box algorithms are less likely to be accepted for use by an organization, and their adoption occurs more slowly than for processes that are directly interpretable [37]. It is useful for a person to be able to understand what manner of policy a robot has learned. It is difficult for humans to trust decisions that cannot be verified, so much research has gone into creating interpretable models [91].

Thus, there is a growing interest in Explainable Artificial Intelligence [13]. Explainable AI can take the form of an AI system that is able to answer specific questions about its policies [43] or a policy that is represented in a more interpretable format [81]. This desire for explainability is a motivation for contributing to this area of Explainable Reinforcement Learning, since many advanced RL methods today result in policies which perform well but which are not directly interpretable.

This thesis progresses along the path towards learning an arbitrary TAIG, via several avenues.

First, we explore transforming a previously learned tabular policy into a decision tree format policy, which we consider an interpretable structure. Decision trees can also be seen as able to encode a subset of the types of instructions that a TAIG can encapsulate. Therefore work on learning trees aids in our eventual goal.

We also extend existing RL methods (pCQI) for learning decision trees and develop our own (CQI and CQI2) in order to enable an agent to autonomously learn a decision-tree-structured policy via RL.

Finally, we propose a RL method that, when certain criteria are met, can learn a full-fledged TAIG format policy.

1.2 Guide to Thesis

Chapter 2 discusses background and related work. Chapter 3 introduces the Transferable Augmented Instruction Graph (TAIG), which is a theory, paradigm, and open-source library that facilitates developing task plans for robots. Chapter 4 showcases real-world demonstrations of TAIG with physical robots, including task transfer. Chapter 5, discusses interactive-TAIG, which enables end-users to teach their robot skills interactively through natural language dialogue. Chapters 6 and 7 discuss using reinforcement learning to allow a robotic agent to learn a task in such a manner that its task plan is transparent and interpretable. Chapter 6 deals with approaches involving human-specified conditions and Chapter 7 is concerned with approaches that determine important features autonomously. Chapter 8 combines elements from the preceding chapters to develop a method to autonomously learn a TAIG that includes two while loops (it requires a task and state/action space formulation that adhere to specific criteria). The common theme is the idea of an instruction-graph-style task-graph as the task driver, object of construction, or means of explanation.

This thesis marks the release of multiple open-source libraries and sample code, including a library for TAIG (Section 3.5) and a new AI-Gym-compatible simulated reinforcement learning environment in which the agent controls traffic lights at an intersection (Section 7.2).

Appendices A and B provide useful resources for using TAIG. Additional information related to the General Purpose Service Robot TAIG demonstration (Section 4.2) can be found in Appendices E and F. Appendix C provides documentation for the Vehicle Intersection AI Gym environment for RL. Appendix D includes information on optimal hyperparameters for CQI2 results as well as what range of values were searched to reach them.

Chapter 2

Background and Related Work

This chapter discusses related work relevant to the efforts, methods, and algorithms proposed and demonstrated in the subsequent chapters of this thesis.

In Section 2.1, we discuss development frameworks and libraries for robots (since TAIG is an open source framework), along with previous explorations into task representation and related ideas. Section 2.2 brings up work relevant to the concept of instructing a robot how to perform a task using verbal monologue, dialogue, or other spoken interaction. Section 2.3 provides background on reinforcement learning and decision trees, two typically unrelated concepts which we will combine in this thesis. Other works that have combined these two areas are also discussed here.

2.1 Development Libraries and Task Representations for Robots

There are a number of open-source libraries targeted at robots. Robot Operating System (ROS) [75], Orocos [21], ROCK [41], and OpenRTM-aist [3] are open source projects that focus on interoperability between the physical and virtual sub-systems that comprise a robot. RoboComp [56] facilitates creating individual inter-operable software components within a robot system. YARP [59] facilitates modularity of devices and components. There are also libraries for specific functionalities such as MRPT [12] for Simultaneous Location and Mapping, MOOS [68] for mobile robots, or Autoware [42] for autonomous vehicles. URBI [6] provides event-driven job execution.

Research exists on automated or assisted plan and policy generation for specific tasks in various contexts [2, 90]. Some planners automatically improve on existing plans [1] or learn a plan by demonstration [5, 24], or through demonstration combined with other forms of feedback [4]. Creating a task through a combination of demonstration and instruction has been explored [70]. Another approach takes advantage of existing knowledge of previously specified tasks [16]. Division of a task plan into high/low-level logic has been explored by representing high level logic as a Markov decision process [82] or other planning methods [22]. The approaches mentioned thus far, however, do not consider task transfer between robots or between very dissimilar tasks.

A natural representation for a graph-representation of tasks may seem to be Petri Nets. Petri Nets can be used for graph-representation of tasks, as in [15, 33]. Petri nets are well suited for describing non-deterministic distributed systems.

Task transfer for similar tasks has been investigated [63]. Other works have trained a model for a problem and then transferred the model to a similar problem [44, 54], or have transferred a model to a different problem in a similar domain [40]. In contrast to these approaches and Petri Nets, our focus is on skill transfer between tasks and task transfer between robots for extremely diverse but well-defined tasks.

Robot Operating System (ROS) facilitates communication between sub-components of a robot [75]. In contrast, TAIG (introduced in Chapter 3) operates on the task level, abstracting the entire task policy itself. (TAIG is fully self-contained, but can be used to complement ROS, as in Section 4.1.3)

Recently, work has categorized aspects of a learned policy as “task-specific” or “robot-specific” [29] . We will apply this idea to tasks that are instructed by an external agent.

The concept of “primitives” (simple actions) in a single robot system is an existing approach [20]. Instruction graph [46], and its extension TAIG (introduced in Chapter 3) are two of multiple possible ways to incorporate this concept.

2.2 Enabling End-Users to Teach Robots Skills by Instruction

There has been much work done on robots Learning from Demonstration [5, 45], where the human demonstrates how to perform a task via a means such as teleoperation and the robot attempts to replicate it. These approaches could be didactic or interactive [51], sometimes being refined after the initial demonstration [49]. Another popular means of transferring task information is imitation learning [7], where the robot views a task being performed and learns how to do it.

In some cases, robots have been created that can combine instructions with previous knowledge [64]. Among other techniques, a robot might determine similar actions that are required between tasks, or that one task is similar to another (differing only in the particulars) [65], or it may retrieve particular demonstration information based on predicted necessary actions [71].

A natural way for a human to give instructions to a robot is using language. Much research has been done on robot understanding of commands or instructions in the context of task learning [26]. Robots have taken advantage of instruction-by-language in the realm of manufacturing tasks [25] as well as in a service robot [38], among other domains. Work has gone into handling uncertainty [38] and variation in spoken human language (as opposed to using a template) [67]. Work has even been done combining such instruction with reinforcement learning, using spoken language as a reward signal to learn anticipated chosen actions [85]. While some studies focus on comprehension of specific commands, others are focused on comprehension through interactive dialogue [50].

In previous work, a robot learned an instruction graph [58]. A specific graph created in each instance was tied to a particular robot (although different graphs were created tied to different robots). In Chapter 5, we will introduce Interactive TAIG, where the robot learns a TAIG. As a TAIG, the graphs created are not system-specific. This method is part of the TAIG framework, is released on the open source library, and can be used with any robot that runs Python.

2.3 Reinforcement Learning and Decision Trees

2.3.1 Reinforcement Learning

Reinforcement learning is a field of algorithms for obtaining policies of (state $\rightarrow$ action) mappings through experience in an environment [83]. We focus on Q-learning: Given a set of states $s \in S$, a set of actions $a \in A$, and an immediate reward for executing action a in a state s, the expected value (Q-value) of performing an action in a state can be learned over time by experimenting with actions in states, observing the reward, and updating the Q-value estimate via the Bellman equation:

$$Q_{t+1}(s,a) \leftarrow (1-\alpha)Q_t(s,a) + \alpha(r + \gamma \max_{a'} Q_t(s',a')) \tag{2.1}$$

where Q_t is the current Q-value estimate for a state-action pair, s and a are the current state and the chosen action, α is the learning rate, r is the immediate reward for choosing action a when in state s (as experienced immediately by interacting with the environment), s' is the next state that the agent is in after executing a, and a' is the best possible next action, such that the maximum expected future reward from being in state s' is added to r after being discounted by the factor γ.

2.3.2 Decision Trees

Decision trees have often been used to create easy-to-understand solutions to classification problems, among others. They are trees that start at a root node and branch based on conditions [76]. An additional benefit to decision trees is that they can be represented graphically, which aids in human understanding [66]. Some work has explored combining neural nets and decision tree forms [84, 94], including using decision trees to explain neural nets [93]. We seek to create a reinforcement learning policy that is in decision tree form.

With the eventual goal of learning a TAIG through RL, we will consider learning a decision tree. The if-else branches in a decision tree can help us determine how to learn the if-else branches in a TAIG.

There are additional benefits to creating a decision tree policy for a task even without a full TAIG structure. Like a TAIG, a decision tree can be understood by a human via visual inspection. In particular, smaller decision trees are of value.

In an AI context specifically, it has been suggested that huge decision trees are insufficiently comprehensible to humans, and that simplifying trees is important *for humans* to interact with the representation [77]. Chess experts had difficulty understanding an early chess-playing algorithm that fully "explained" itself with a complex decision tree–too complex a tree becomes opaque [60, 61]. It has been shown that, all else being equal, humans prefer simpler explanations compared to more complex ones [52]. Based on the foregoing, we assume for the remainder of the thesis that smaller decision trees are more interpretable by humans.

2.3.3 Learning Decision Tree Policies with RL

We are not the first to attempt to learn a decision tree policy. The "G algorithm" [23] learns a decision tree incrementally as new examples are added. The Lumberjack [88] algorithm is for a Linked Decision Forest – like a decision tree but without the need for repeated internal structure which would otherwise occur. TTree [86, 89] solves MDPs or SMDPs and involves both state abstraction (grouping states together and treating them as a single state) and temporal abstraction (referring to techniques that group sequences of actions together and treat them as one abstract action). [74] is a work which uses a tree as an internal structure and incrementally builds the tree, splitting on Q-values. Unlike [36], an approach that uses decision trees as part of the representation, we use a single decision tree to represent the entire policy. [92] creates decision trees using a post-learning transformation. In our approaches, we instead maintain our policy as a tree at all stages of training and merely update our tree.

Structured Policy Improvement [17, 18] represents state as a series of boolean conditions, and uses a tree as an internal structure (although not as the final policy representation). Also of note is that this method does not start with random sampling but with an existing tree initialized with a policy resulting from greedy one-step evaluation (learning a policy without taking into account future reward).

Another approach to decision tree policy policy via RL is found in [92], but that approach still requires the backing of a table-like structure from which the tree is built, requiring maintaining information about each specific visited state.

The UTree algorithm [57, 87] learns a tree, starting with a single abstract state and then continually splits as appropriate (based on a splitting criterion), ending when a stopping criterion is reached. U-Tree based algorithms split a leaf node when there are enough historic Q-value updates (when the mean is less than 2σ and a minimum number of samples are present in a leaf's history list). This is done regardless of whether the policy would change for either of the new leaves, so unnecessary splits occur. Additionally, only on-policy transitions are used to populate history lists. As a result, an initially pessimistic estimate for an action's Q-value can prevent it from being chosen and therefore prevents its Q-value from being updated. This can prevent an agent from converging to a better policy. An effort to produce a decision tree via RL is found in [34], but the final product is not directly interpretable: it consists of a tree with linear Gibbs softmax sub-policies as leaves. In contrast, we seek approaches that create a tree with action choices as leaves.

VIPER distills a deep neural network policy into a decision tree policy for the purpose of performing validation and verification on the decision tree format [10].

The Pyeatt Method [73] (PM) starts with a single root node and adds branches and nodes over time. PM maintains a history of changes in Q-values for each leaf, and it splits when it believes that this history represents two distributions. We will use PM as a reference method in Chapters 7 and 8.

2.4 Summary

This chapter has presented related work and background information on a number of topics useful for understanding the proposals and discussions in the chapters to come. The work on TAIG is a continuation of instruction graph, and it fits into the field of task representation. With the release of the library, it joins a number of other open source robot libraries. Building upon existing instructional and conversational dialogue work, we will demonstrate a means to use an interactive dialogue to construct a TAIG. CQI and CQI2 are reinforcement learning methods with decision-tree style policies that build on prior work in the RL and decision tree space. Learning TAIGs with RL builds upon much the foregoing research and the rest of the thesis itself.

Over the course of this thesis, there will be both learning from humans and autonomous learning. The vision of this thesis entails a combined human-machine learning paradigm.

Chapter 3

Transferable Augmented Instruction Graph: Formalization

This chapter describes and formalizes the Transferable Augmented Instruction Graph (TAIG). TAIG is a task representation that extends and improves upon Instruction Graph (IG) [58].

A TAIG can handle more complex and demanding tasks than a simple IG, making it more suitable for use in the real world. State and environment information can now be tracked and recorded. Sub-tasks can be defined and halted.

Significantly, TAIG makes it easier to design and execute transferable tasks for robots, potentially accelerating the introduction of useful robots into society at large. TAIG allows a robot that knows one task to re-use existing knowledge where possible. The method further allows this knowledge to be shared between disparate robots, despite physical and software differences.

TAIG allows for executing the same instruction set across multiple different robot platforms. The task logic is abstracted from the specific attributes of each robot.

We first discuss the "Augmented Instruction Graph" (AIG), an enhanced version of an IG that aims to model and execute more complex tasks than could a simple IG.

We then place AIG in the context of a modular architectural paradigm that makes it transferable. We call this "Transferable Augmented Instruction Graph" (TAIG) since it aims to enable us, for the first time, to use identical IGs on different robots.

In order to enable our techniques to be used by the community, we have created an open-source library and released it.

This chapter is organized as follows: Section 3.1 introduces the Instruction Graph as background work. Sections 3.2 and 3.3 describe the approach taken to extend the IG and make it transferable, with additional augmentations discussed in Section 3.4. TAIG's ability to handle more complex tasks than a simple IG and enablement of transferability is demonstrated in Chapter 4.

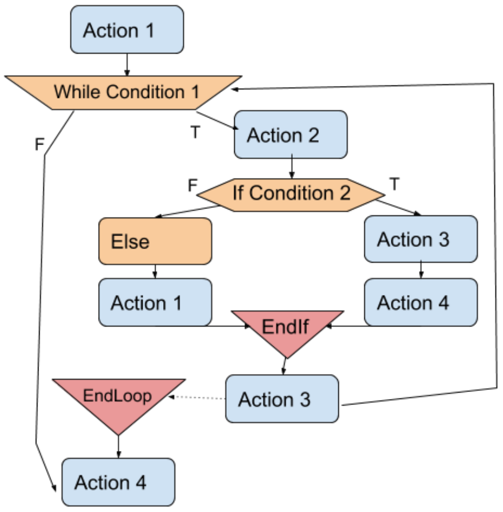


Figure 3.1: The structure of an example IG.

A tutorial for using TAIG with a Pepper robot is found in Appendix A and developer-oriented documentation for the TAIG library is found in Appendix B.

3.1 Background: Instruction Graph

The concept of "primitives" (simple actions) in a single robot system is an existing approach [20]. In previous work, the concept of representing the steps of a task as an Instruction Graph [58] was developed. Instruction Graphs are directed graphs, where each vertex node represents a small unit of work or a condition to test. These units are called "primitives." Edges between nodes indicate how the graph can be traversed. Formally, each node v can be represented as

$$v =< \text{id}, \text{InstructionType}, f, P > \tag{3.1}$$

where id is a unique node id, `InstructionType` is the type of node, f is the literal function to be executed when that node is run, and P are the parameters to pass to f. Types can be "Do" (performs an action), "If Condition" (tests an if-condition, affecting which edge to traverse next), and "Loop" (runs a series of nodes while a condition is true), among others. (See a graphical representation of an example of this kind of structure in Figure 3.1.) Edges can be represented by

$$e =< v_i, v_j > \tag{3.2}$$

indicating a progression from node v_i to node v_j.

Each Instruction Graph has a single beginning and ending node. There can be multiple paths through the graph. Execution traverses the directed edges through nodes, executing each function f therein. If a node has multiple outgoing branches, the output of f determines which edge to traverse.

The tuple in Equation 3.1 is hereafter referred to as a Node Tuple (as distinct from the Primitive Tuple introduced later in Section 3.5).

Further work on IG has focused on adding an “autocomplete” to speed task creation [32] and on the use of IGs in multi-robot scenarios [46]. This thesis addresses the challenge of efficiently creating dissimilar tasks with shared components, allowing for interactive graph creation, and enabling multi-robot task transfers.

Subsequent sections of this chapter will change the definition of the node tuple and expand on IG to make it more robust and transferable. New features will be added that make it suitable for general use and incorporate instruction graph into a larger architectural paradigm.

3.2 Augmented Instruction Graph

3.2.1 Memory

The first of the significant contributions of this chapter is the addition of a concept of a central memory that gets used by the AIG during execution.

In a simple IG, primitives (actions the robot can perform) have no conception of state, of history, or of any other sort of environmental or temporal context. Task actions are limited to reactions to the present environment, as opposed to reasoning about the future.

We might also want the controller of a robot to have information about a network connection or other session-based information (as in our demonstration in 4.1.2). We may want the primitives to have access to database connections or other streams of data. Primitives alone, being stateless, atomic actions or condition checks, could not handle this scenario.

In a simple IG, some of this behavior could potentially be programmed if one made assumptions about the underlying robot system. However, it would also be desirable to be able to use our primitive functions with any backend (database, file system, local computer memory, or even connections to networked data sources), to enable transfer between robots, and to upgrade a robot system without breaking the task.

We give an AIG the ability to have access to a consistent read-write memory active during the runtime of a task, allowing the robot to keep track of the above-mentioned information.

This central memory is managed by an object called a Memory Object. A Memory

Object has an arbitrary set of named attributes, to or from which arbitrary types of data can be written or retrieved. Before the execution of an Instruction Graph, a specific Memory Object to be used for that execution is specified. (No Memory Object needs to be specified during graph creation, although the user should have one in mind.)

In Equation 3.1, a primitive function and arguments to pass to it are specified for each node. In addition to any arguments specified in the P list, a “Memory Object” (associated with the current execution) is passed to the primitive function as the first argument (preceding the parameters in P). (See Figure 3.2 for visual diagram.) The function can then use this object directly and use its methods to read and write information to memory.

By providing this functionality to the primitives at runtime, an AIG gains the ability to have memory across the execution of the task. This allows for keeping track of state and historical records.

A primitive will not interact directly with the memory of the system it runs on. Primitives’ functions don’t have to care about what sort of backends are used. The primitives determine what to read and write, and the Memory Object provides and stores the data. The Memory Object will take care of any requisite backend memory operations for the specific robotic system for which it was created. The primitives themselves are still stateless but the task as a whole has memory. (Each primitive has access to that memory when it executes.)

It is not required for an agent to explicitly instruct a robot to save information. For a hypothetical task that involve remembering a person’s name and face, primitives could be created to cause the robot to view the person, listen for their name, and save this information to memory according to the instruction, “Remember the person you are looking at.”

A Memory Object can also be used to provide initial state, environment, or other robot-specific information to a robot. For example, the controller of a robot may require information about a network connection or other session-based information in order to execute commands. In this scenario (which is applicable to our demonstration later), it would not even be possible to execute an IG running primitives on this robot without the use of an AIG’s Memory Object to handle the maintenance of session information. Similarly, a Memory Object can be used to open and keep open any database connections or streams that primitives would then be able to access and utilize (and to which they could write and from which they could read).

Further, one can easily change environment information by modifying it once in the Memory Object, which will seamlessly provide it to all primitives.

3.2.2 Negation

In a simple IG, if one desired to create a graph that contained one or more tests of when a condition was true and also tests of when that condition was false, it would be necessary to create *double* the number of primitives for every such case.

In our approach, the same primitive can be used in both cases because negation of conditional nodes is allowed. This is most applicable to LOOP nodes. The Node Tuple is modified from Equation 3.1 to the following:

$$v =< \mathrm{id}, \mathrm{type}, \mathrm{pid}, P, \mathrm{neg} > \tag{3.3}$$

where `id` is a unique node id, `type` is the type of the node, and neg is a boolean field indicating negation. Instead of having a literal function f there is a primitive reference `pid` (discussed later in Section 3.3.1). P are parameters that will be passed to a function stored on the primitive referenced by `pid`.

For an action tuple node the `neg` attribute is ignored. For a condition tuple node (such as IF or WHILE), if `neg` is set to False, the node is traversed as normal. If `neg` is set to true, then the condition result is flipped. For any conditional node,

$$\mathrm{node\_result} = \mathrm{cond\_fun\_result} \oplus \mathrm{neg} \tag{3.4}$$

where `cond_fun_result` is the boolean result of the function specified by the primitive on the node at the time of execution, `neg` is the node tuple field, and `node_result` indicates which branch of the graph should be subsequently traversed.

3.3 Transferable Augmented Instruction Graph

Now that an AIG can be created with memory and negation, it is possible to represent as complex a procedure as needed for many real-world tasks. The following sections will now discuss how to make the AIG into a TAIG, which is transferable across robots.

3.3.1 Primitives and Primitive Library

It is desirable to have graphs be serializable, so that they can be saved to a file and used across systems. Previously, with functions existing directly on node tuples, this meant that either the function had to be serializable as well, or else a string of code to evaluate would have to be stored (a bad coding practice).[1] However, it is undesirable to have any of these restrictions on what functions can or cannot be used as primitives.

[1]Using `eval` or `exec` or similar constructs in any language is typically slower, insecure, and harder for future developers to debug [8, 11, 72].

It would also be useful to be able to use different sets of primitives for different robots. If a task requires communicating with a user, for example, it would be ideal to have the same type of node in the graph represent "get input from user" and "announce output to user" regardless of the robot. However, the implementation details would be different depending on the robot. A robot that could speak and listen to words would communicate with a user by this method, while a robot that could do neither might instead display information on a screen and take input in typed text form. These robot-specific details should be decoupled from the graph representation, which should be concerned only with how to execute the task itself.

We also want to re-use primitives wherever possible. A robot should be able to use existing skills for new tasks, and task-specific code should not be tied to a specific platform, but be able to be itself re-used across robots. TAIG enables this clean separation between robot- and task-specific primitives. A system without a formal concept of atomic functionalities may have the transferable and particular content intermixed and more tightly coupled.

Previously, the functions that would be stored in a Node Tuple would just be raw functions existing in whatever was the namespace in which the IG was created. (The previous three paragraphs illustrate the problems with this approach in the areas of serializability, interchangeability, transferability, and re-usability.) This paper introduces the Primitive Library, which is a collection of primitives. TAIGs are built out of these smaller primitives.

When a user (or robot agent) is creating or running an Instruction Graph, a Primitive Library will be associated with this process.

The primitives themselves are now tuples. Formally, a Primitive Library `PL` is a set of Primitive Tuples `PT` ($\texttt{PT}_i \in \texttt{PL}$):

$$\begin{gathered} \mathrm{PT}_i =< \mathrm{pid}, t, f > \\ t \in \{\mathrm{Action}, \mathrm{Condition}\} \\ \text{s.t. } \mathrm{PT}_i[\mathrm{pid}] \neq \mathrm{PT}_j[\mathrm{pid}] \quad \forall i \neq j \quad \text{where } \mathrm{PT}_i, \mathrm{PT}_j \in \mathrm{PL} \end{gathered} \tag{3.5}$$

$$f = \begin{cases} \mathbf{f} : P \to \emptyset & \text{if } t = \text{Action} \\ \mathbf{f} : P \to B & \text{if } t = \text{Condition}, \quad B \in \{\text{True}, \text{False}\} \end{cases} \tag{3.6}$$

where `pid` is the primitive identifier, t is the Primitive Tuple type, f is function that can take parameters, and P indicates an arbitrary number of specific arguments ($P = \emptyset$ is allowed). In the node tuple on a graph in Equation 3.3, the `pid` corresponds to a `pid` on a Primitive Tuple in the associated Primitive Library. When the node is executed, the primitive is looked up by reference, and the parameters P are passed.

All node types can be classified as taking either an Action Primitive (for example, an "Action" / "Do" node), a Condition Primitive (for example, an "If" node), or no primitive (for example, "End If" and "End Loop" nodes).

Serializing a graph to a file is achievable now since the node tuple now stores a reference (string) to the primitive tuple (which contains the actual function). Since `pid` is always serializable, and only the node tuples (and not primitive tuples) are saved in the instruction graph file on disk, there are no longer any restrictions on what types of a functions f can be.

The primitives have been decoupled from the nodes. The graph representing the task plan is decoupled from the robot-specific implementation. A `pid` is unique within a `PL`, but different `PL`s may have different functions for the same `pid`. (A `pid` may be re-used within a graph of course, to refer to the same primitive multiple times—that is the purpose.) Any function can be a primitive now, since what is stored in the node tuple now is simply the reference to the primitive tuple `pid`. The restrictions inherent in the simple IG structure are eliminated.

Additionally, a human or robot TAIG creator does not require access to a fully implemented or loaded-in-memory primitive to create an instruction graph. (This could help with development on a large system.)

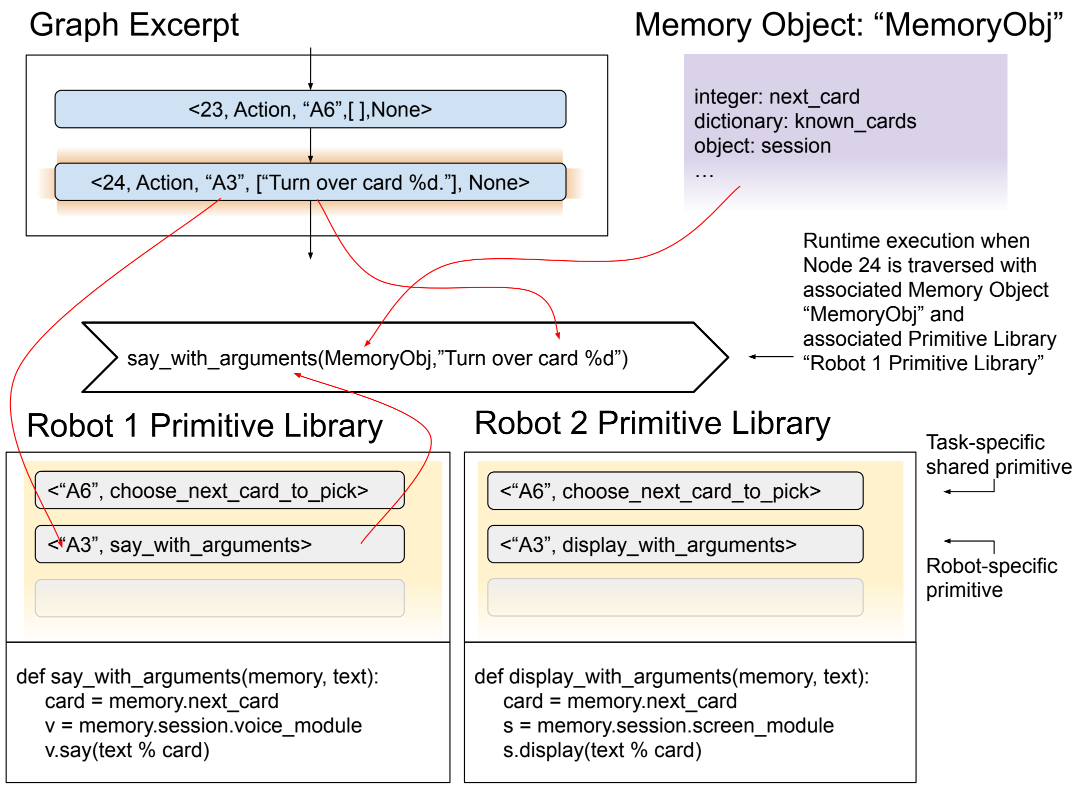
We can use TAIG to use one graph to perform a task with multiple different robots (as is demonstrated in Section 4.1), by swapping in the library meant for a new robot. We can use multiple libraries with the same robot as well.

3.3.2 Bringing it All Together: The Library-Memory-Graph Paradigm

With TAIG, the task policy can be represented independently of a robot, in contrast to platform-specific approaches. Additionally, the graph itself can be directly executed by a robot and transferred between robots without modification. (This is in contrast to a form that is just a description of some machine-readable-only format.)

A system that creates or executes a TAIG utilizes three previously discussed components: i) the Primitive Library, containing atomic actions and boolean condition checks, ii) the Memory Object, providing an abstracted interface to read-writable memory, and iii) and the Graph, describing the high-level task logic.

In an example scenario showing an interaction between these components in Figure 3.2, the robot is playing a card game, where there are numbered cards on a table. Previously, in node 23, the robot has determined to pick a certain card. Now, traversing the graph to node 24, it reaches a node whereby the robot tells the user the number of the card it has picked. Depending on which robot is running the TAIG, a different means of communicating the card number to the human is used. In this example, Robot 1 Primitive Library is associated with the graph, so the robot uses voice. Also shown are two primitives from an alternate PL for Robot 2, that tells the user the card via a display on screen instead of via voice. In both cases, the task graph is the same, and the card number is retrieved from memory. Contrast this to


Graph Excerpt
<23, Action, "A6",[ ],None>
<24, Action, "A3", ["Turn over card %d."], None>
Memory Object: "MemoryObj"
integer: next_card
dictionary: known_cards
object: session
…
Runtime execution when Node 24 is traversed with associated Memory Object "MemoryObj" and associated Primitive Library "Robot 1 Primitive Library"
say_with_arguments(MemoryObj,"Turn over card %d")
Robot 1 Primitive Library
<"A6", choose_next_card_to_pick>
<"A3", say_with_arguments>
Robot 2 Primitive Library
<"A6", choose_next_card_to_pick>
<"A3", display_with_arguments>
Task-specific shared primitive
Robot-specific primitive

```
def say_with_arguments(memory, text):
    card = memory.next_card
    v = memory.session.voice_module
    v.say(text % card)
```

```
def display_with_arguments(memory, text):
    card = memory.next_card
    s = memory.session.screen_module
    s.display(text % card)
```

Figure 3.2: The three components of the Memory-Library-Graph paradigm interact.

the graph shown in Figure 3.1, where the primitives are integrated into the graph instead of decoupled, and the independent Memory does not exist.

In the software, this relationship is managed by a "Manager". At a single point in time, the Manager holds one Primitive Library, either one Memory Object or no Memory Object, and either one complete TAIG, no TAIG, or a TAIG in the process of being constructed. The human or machine agent initializes the Manager object with a Primitive Library and, optionally, a Memory Object. These can be changed for other Primitive Libraries and Memory Objects later, between executions of a TAIG (including between executions of the same TAIG). The Manager begins without a TAIG. The agent can choose to i) begin creating a new TAIG from scratch or ii) load a TAIG. After a TAIG is loaded, the agent can choose to i) add to the TAIG or ii) execute the TAIG. After creating or modifying a TAIG, an agent can choose to i) save the TAIG with a new or the same name, or ii) execute the new TAIG.

In this manner, the methodology above achieves a modular architecture for creating and utilizing instruction graphs. There is a separation between memory, task logic, and atomic task primitives. From a developer perspective, there is an opportunity for reduction in duplicate code.

Under this new paradigm, the same graph can be run on multiple robotic systems. Many primitives can be shared between libraries, decreasing the cost in time, effort, and lines of code required to implement a task on a new robot.

One of the previous limitations was that an IG would have to be executed in the same environment as that in which it was written. Now, the TAIG can be used in multiple environments unchanged. (The Primitive Library counts on a specific interface to exist in the Memory Object, eliminating any need to have a primitive refer to a specific system directly and be thus tied to that system.)

If a team is using different databases or sources for development, testing, and production use, different Memory Objects could be employed to easily enforce this separation while using the same code for the actual task execution.

For transferring a task from one robot to another, primitives that are task-specific but not robot-specific can be re-used, while only robot-specific primitives must be re-entered. For implementing an additional task on the same robot, primitives that are robot-specific but not task-specific can be re-used, while only the truly new functionality for the new task must be implemented.

For a properly constructed TAIG, transferring a task from one robot to another does not require modifying a TAIG at all. The exact same TAIG can be re-used.

3.4 Further Augmentations

3.4.1 Nested Graphs (Graphs as Primitives)

TAIGs can be saved to and loaded from files. Previously, they could be saved, loaded, and run only in their entirety.

We have added an included-by-default **run_ig** primitive, which appears in the Primitive Library without being explicitly defined.[2] The **run_ig** primitive is an action primitive and can be used anywhere an action primitive can be used. It takes a single optional string parameter (the name of the instruction graph to run).

There are two ways to use the **run_ig** primitive. First, explicitly specify the graph to run. Second, specify no argument. In the latter case, during execution the primitive will attempt to retrieve the name of the graph to run from the memory object and then run it. Of course, this requires creating and running a primitive that will set this name in the memory object ahead of time.

This possibility for dynamic execution is very useful in situations where pre-specified subtasks are known, but the manner in which they are to be combined or executed is not.

[2]If a developer were to create their own custom primitive with the same string id as the **run_ig** primitive, the **run_ig** primitive would dynamically choose a new string as a unique id (and this id can also be retrieved programmatically).

The child graph itself is executed using the same Manager, Memory Object, and Primitive Library as the parent graph. The child graph runs from beginning to end, and then the execution returns to the parent graph. (The only exception to this is noted in Section 3.4.2.)

A child graph can itself run another child graph, drawing from the same set of instruction graphs. There is no limit as to how deeply graphs can be nested, aside from limits imposed by the physical memory of a machine.

3.4.2 Halt Conditions

We introduce the concept of a *halt condition* in an instruction graph. The halt condition serves as a virtual kill-switch. This has applications in the areas of safety and control.

During the construction of a graph, in addition to the graph structure, one can specify a condition primitive as a "halt condition."

During execution, if a graph has a halt condition specified, the condition is checked prior to the execution of each node. If the condition evaluates to true, the entire graph halts execution immediately.

In addition to safety, this is useful in terms of application flow when combined with nesting graphs. If a child graph (a graph run by the **run_ig** primitive) terminates execution due to a halt condition, the control flow returns to the parent graph in the same way as if it had completed successfully. This is useful for creating task procedures using instruction graphs that include handling unexpected conditions or otherwise escaping scenarios.

3.5 Open Source TAIG Library for Use by the General Robotics Community

This thesis marks the debut of an open source library for TAIG. The library is available on pypi[3] and can be installed into any Python environment with the command

```
$ pip install instruction_graph
```

The `instruction_graph` library can be used by anyone to use TAIG in their own projects. It can be used on any robot or other system that can run python. It has been tested and is compatible with both Python 2 and Python 3. The source code along with documentation is available on github.[4]

[3]Instruction Graph library available as a pypi Python package: https://pypi.python.org/pypi/instruction-graph

[4]https://github.com/AMR-/instruction_graph

Although code for previous robot-and-task specific instruction graphs has been released before, this marks the first general-purpose instruction graph library suitable for general use, specifically designed for use by the public. It is also, of course, the first to feature TAIGs and the Memory-Library-Graph paradigm.

All of the TAIG-related demonstrations and examples described in this thesis utilized this library to use TAIGs to create and execute tasks.[5]

3.6 Summary

This chapter introduces the formal definition of TAIG along with an open source library to facilitate its use. Building on IG, TAIG contains memory, negatable conditions, nested graphs, and halt conditions. This allows for representing tasks of a higher level of complexity than the IG. TAIG also entails primitives decoupled from the graph in the form of the Primitive Library. The Library-Memory-Graph paradigm enables transfer between robots as well as other benefits. Chapter 4 walks through a few demonstrations of TAIG in the real world.

[5]Specifically, `instruction_graph` version 0.1.9 was used in Section 4.1 and 0.2.23 was used in Section 4.2.

Chapter 4

Transferable Augmented Instruction Graph: Demonstration

Chapter 3 discussed the theory of TAIG in detail. This chapter shows real-world examples of robots running TAIG.

The demonstrations in this thesis run TAIG on commercially available robots in the physical world. Section 4.1.2 demonstrates on a Softbank Pepper robot how TAIG can be used to perform a task that could not be accomplished with a simple IG. Section 4.1.3 shows the benefits of the TAIG system in terms of inter-robot compatibility, as the same task (a card matching game) is easily transferred from the Pepper robot to the Rethink Robotics Baxter robot. The usefulness of halt conditions and nested graph capabilities is illustrated in a further demonstration in Section 4.2, with a home-assistant challenge task (called "GPSR").

4.1 Demonstration and Qualitative Comparison: Task Transfer

First, the Softbank Pepper robot is used to explore the ability of a TAIG to execute a task that a simple IG would not be able to perform. Then, it is demonstrated how the **exact same TAIG** can be executed to perform the same task on a completely different robot, the Rethink Robotics Baxter.

4.1.1 Robot System

Pepper and Baxter are extremely different robots. Baxter has arms meant for dexterous manipulation, is stationary, and has a tablet display. Pepper is explicitly reminiscent of a human, with arms meant more for gesticulation than manipulation, and the ability to move around. Pepper can voice words, while Baxter has no audio output. The camera sensors are very different as well. Baxter runs the Ubuntu operating system and interfaces with other systems via ROS, while Pepper runs a custom version of Linux called NaoQi.

4.1.2 Task Demonstration: Pepper Matching Game

The task which will be used to demonstrate the new TAIG paradigm is Pepper playing a matching card game. Pepper and a human player take turns picking two face down cards in sequence. If the cards are revealed as a pair when turned over, they are removed from the board. If not, the cards are turned back face down. The last player to remove a pair wins the game.

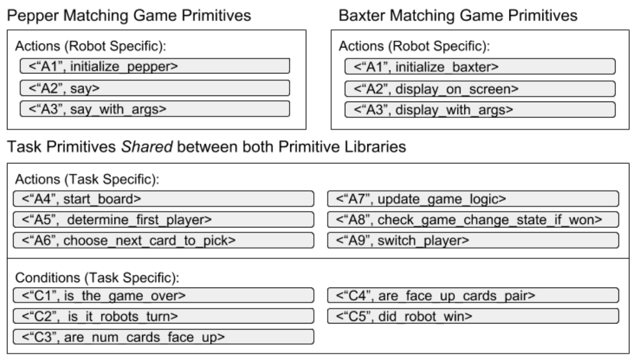
The graph for this game is shown in Figure 4.1b and the primitive library in Figure 4.1a. Note how 27 out of 33 (81%) of the nodes in 4.1b use a primitive that is re-used multiple times. In all of these instances, the TAIG is saving memory and complexity by referencing the primitive tuple using a string key instead of storing the function directly.

At multiple points in the game, Pepper must wait while the human turns 1 or 2 cards face up or face down. To enable implementing the TAIG, a primitive was made that takes one integer argument, and returns true if that number of cards are face up. By referencing this primitive in a Loop node that is negated, the useful functionality of *waiting until* that number of cards is face up is achieved.

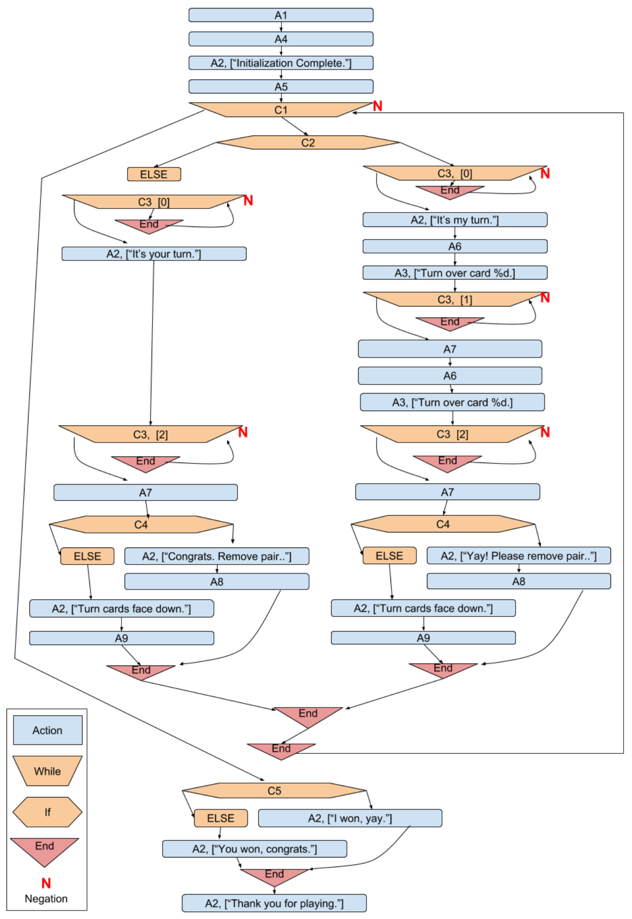
Memory is essential to task performance. Without the capability to store information about which cards are where, it would not be possible to win. The Memory Object allows us to track information over time.

The Memory Object is also used in this case to maintain session information about the robot. When writing a program that interfaces with the Pepper robot specifically, it is a prerequisite to keep track of this information (or re-acquire it). This session information is used to obtain sensor readings and execute lower level motion or audio commands. If this session information were not provided to the primitives by the Memory Object during execution, it would be required to re-initiate a new session at every step. This would be less efficient than our method, which initiates the session once and re-uses it throughout execution, providing it to the primitives as needed. The primitives related to Pepper readings and actions do not care about opening or closing this connection, they simply use it. The primitives continue to be stateless and atomic themselves, but the application as a whole has memory.

The graph itself contains no robot-specific information. This will enable the task to be transferred to a different robot, as described in the next section.


Pepper Matching Game Primitives
Actions (Robot Specific):
<"A1", initialize_pepper>
<"A2", say>
<"A3", say_with_args>
Baxter Matching Game Primitives
Actions (Robot Specific):
<"A1", initialize_baxter>
<"A2", display_on_screen>
<"A3", display_with_args>
Task Primitives *Shared* between both Primitive Libraries
Actions (Task Specific):
<"A4", start_board>
<"A5", determine_first_player>
<"A6", choose_next_card_to_pick>
<"A7", update_game_logic>
<"A8", check_game_change_state_if_won>
<"A9", switch_player>
Conditions (Task Specific):
<"C1", is_the_game_over>
<"C2", is_it_robots_turn>
<"C3", are_num_cards_face_up>
<"C4", are_face_up_cards_pair>
<"C5", did_robot_win>

(a) The Primitive Libraries for the "matching game" task for Pepper and Baxter. Three Primitives are different and the remainder (the majority) can be shared between the two Primitive Libraries without modification.


A1
A4
A2, ["Initialization Complete."]
A5
C1
N
C2
ELSE
C3 [0]
N
End
A2, ["It's your turn."]
C3, [2]
N
End
A7
C4
ELSE
A2, ["Congrats. Remove pair.."]
A8
A2, ["Turn cards face down."]
A9
End
C3, [0]
N
End
A2, ["It's my turn."]
A6
A3, ["Turn over card %d.]
C3, [1]
N
End
A7
A6
A3, ["Turn over card %d.]
C3 [2]
N
End
A7
C4
ELSE
A2, ["Yay! Please remove pair.."]
A8
A2, ["Turn cards face down."]
A9
End
End
End
C5
ELSE
A2, ["I won, yay."]
A2, ["You won, congrats."]
End
A2, ["Thank you for playing."]
Action
While
If
End
N
Negation

(b) The TAIG for the "matching game" task. Argument lists are only shown on nodes if they are not empty. Negation boolean is indicated if it is true, assumed to be false otherwise.

Figure 4.1: Primitive Library and Task Graph for Matching Game Task

4.1.3 Robot-Independent Task Transference: Baxter Matching Game

The new Memory-Library-Graph paradigm enables us to take the TAIG generated for Pepper and execute it–***unchanged***–for Baxter. Previously, a specific IG was tied to a particular robot. Instead, in our case, we simply saved the graph to a file and loaded it to execute on Baxter. All that was required was implementing a new Memory Object, and implementing the minority subset of primitives that were robot-specific.

The Baxter and Pepper Memory Objects kept track of game logic using the same fields. There were a couple of robot-specific differences. The Pepper Memory Object kept track of a Pepper session, which is irrelevant to Baxter. Baxter uses ROS[1] and so the Memory Object for it set up subscribers and publishers to certain topics.

Next, new primitives were implemented as necessary. Out of 14 primitives, 11 could be re-used and only 3 had to be implemented anew. The two primitives related to communicating with the human ("A2" and "A3" in Figure 4.1b) had to be re-implemented. In particular, Pepper can speak with words, while Baxter has no voice capability. In the case of Baxter, the robot would display words (or images) on its screen. The initialization primitive was the only other primitive to require new code, since initialization is different for these two disparate robots.

The means of attaining images from the camera changed as well, from an API call to a ROS topic. The image dimensions and image formats also differed. Despite these differences, no change was required in any primitives in order to handle camera input. This was taken care of by i) the Memory Object, which in each case retrieved the image from the appropriate location and stored it in a common format expected by the primitives that would retrieve it later, and ii) the initialization primitive, which set up and stored (in the memory) the particular calibrations for the camera used on each robot. (The image dimensions differed between robots too, but our code to analyze and determine card information from an image was robust to such changes.)

The majority of the code is unchanged.

Previously, implementing an existing task on a different robot, even with an IG, would likely have required writing the program mostly from scratch, perhaps with some manual copy-and-pasting from existing code. In a simple IG, keeping track of state would have had to been done by a primitive assuming that it was running on a particular machine and interfacing with that robot directly.

With TAIG, the game-logic related primitives can be run on any machine, not just the one for which they were created. The Memory Object determines the appropriate location in which to store the information. The Memory Object allows primitives to be invariant as to whether the data is stored on a Baxter, on a Pepper or on an

[1]It is possible for Pepper to use ROS as well, but in our implementation of Pepper playing the matching game ROS was not used

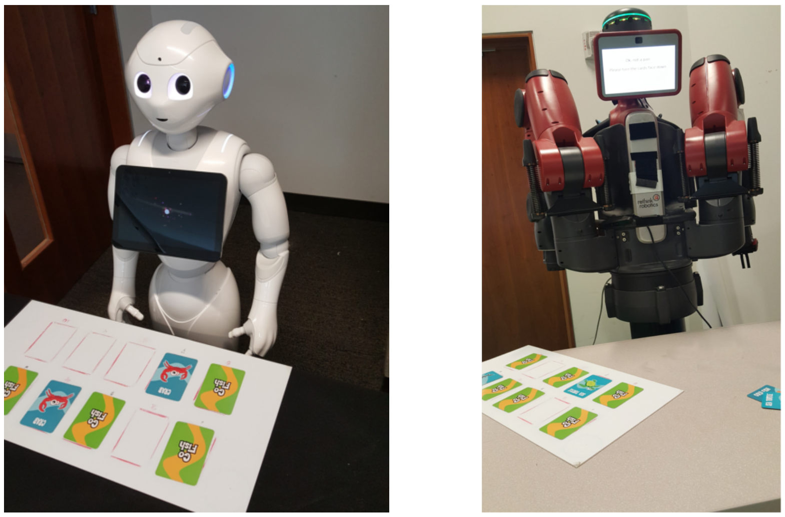


Figure 4.2: Pepper and Baxter play the matching game.

external device.

In Figure 4.2, Pepper and Baxter are each shown playing the matching game. Baxter's screen indicates that it has found a pair, while Pepper is voicing this fact aloud. A video of a segment of this game can be found at `https://github.com/AMR-/CMU-Structured-Representations-For-Robots-Videos/blob/master/Transferable_Augmented_Instruction_Graph_Video.mp4`

As shown, in the Memory-Library-Graph paradigm, once the task logic is specified, all that requires being changed are the particular atomic functionalities that are truly robot-specific. Task handling related to the task itself (if properly separated into task and robot primitives) largely does not need to be rewritten. The task-based, robot independent primitives can be used without modification.

An identical set of instructions was used to perform the same task on different robots. Handling different types of hardware is accounted for, and even different capabilities (speech abilities on Pepper compared to lack of a speaker on Baxter) can be addressed. The specific low-level instructions differ. A minority of robot-specific actions differ. The Instruction Graph itself, however, is identical across both robots. The very Instruction Graph that was generated for Pepper playing the matching game was saved to a file. Using the exact same Python commands on Baxter as on Pepper to load and run the TAIG, this file was loaded and was executed without modification, causing Baxter to play the matching game as well.

4.2 Practical Demonstration: General Purpose Service Robot

RoboCup@Home is an international competition for at-home robots. This section discusses how TAIG helped enable our Pepper robot to tackle a task such as General Purpose Service Robot (GPSR).

4.2.1 RoboCup@Home and the General Purpose Service Robot Task

RoboCup@Home (www.robocupathome.org) is an international competition in which robots perform tasks meant to imitate activities that a household assistive robot might be expected to perform, such as retrieving groceries, guiding a guest through the house, or fetching drink orders at a cocktail party. One of its leagues is the Social Standard Platform league, in which Pepper robots are used.

One of the most challenging tasks in the first stage of the competition in 2018 is called "General Purpose Service Robot" (GPSR).

In GSPR, the robot will receive three commands (one at a time). Each command is generated randomly from a grammar. There are millions of possible sentences that can result. Find examples listed in Figure 4.3.

Tasks may involve numerous robot capabilities including moving to locations, searching for objects or furniture, identifying objects, identifying people, counting people and objects, finding people, following people, guiding people, pose detection, gender detection, gesture detection, clothing detection, carrying objects, and dialogue-based interactions with humans (to learn information, deliver a message, or for other purposes).

This test is presented as a case study since it highlights the utility of the Task Representation research that was used to address the test.

4.2.2 Robot System

We use the Pepper robot. In addition to the TAIG framework, we use an object recognition system and NLP framework.

Object recognition is accomplished through a CNN with data augmentation techniques based on [31] and trained with YOLOv2 and pre-trained weights darknet19_448.conv.23 for the convolutional layers.

The Robot has an NLP framework developed in the CORAL lab (a successor to that described in [27]) that can listen for audio with Pepper's microphones and transform the audio into text using either Pepper's inbuilt Nuance Vocon or Google

- navigate to the living room find a waving person and say what day is today
- tell me the gender of the person at the entrance
- tell me the name of the person in the living room
- tell me how many people in the dining room are boys
- find the lightest cutlery in the kitchen
- tell me how many fruits there are on the side table
- pick up the grape juice and place it on the storage table
- bring me the orange
- deliver the fork to the person raising their right arm in the corridor
- navigate to the counter locate the paprika and deliver it to robin at the desk
- follow robert from the bed to the corridor
- lead charlie to the bookcase you will find them at the couch

Figure 4.3: Example GPSR Category 2 Commands

Cloud Speech. Each recognized utterance is parsed using a linear chain conditional random field parser trained on the specific task and subtask in question.

Finally, each parsed utterance is matched with a parametrized **frame** that represents a robot command. At a given time, the parser is attempting to match to a specific set of frames, called a **frameset**. The parse with the best match is returned if it contains the required frame parameters.

The frame is provided to the application on a ROS topic with a ros message that has 6 fields aside from the header:

- **frame** - a string identifier for the frame (e.g. "Deliver Object", "Find Object In Room", "Drink Order", "Yes")
- **raw** - a string of the raw text received
- **parameters** - a dictionary of key-value pairs representing parameter names and values (e.g. "room" $\rightarrow$ "living room")
- **incomplete** - a boolean to indicate whether the frame is "incomplete". If false, it means that the parsing was completely successful. If true, it means that the parsing encountered some errors, but that there was enough information to indicate *something* (and it is up to the application to choose how to handle this partial information, if at all). If and only if **incomplete** is true, one or both of the **missing** and/or **unexpected** fields will be populated (if false, neither will be)

- **missing** - a string array of parameter names that were missing from the sentence
- **unexpected** - a dictionary of key-value pairs of parameters whose values do not correspond to the expected parameter type (e.g. "room" → "pasta")

We developed a Memory Object to use with GPSR that listens to this ROS topic and stores the most recent frame.

There are primitives in the Primitive Library used here which set or change the frameset, and another that consumes a frame.

To give an example of a frame, consider the command, "find the largest apple in the bedroom." The parser will recognize this as the frame "findobjcatextreme" (Find Object/Category by Extreme-Attribute). The "findobjcatextreme" frame has three parameters. First, the attribute ("largest"), the object or category ("apple") and the room ("bedroom"). The resulting ROS message published would have the following structure:

`frame` → "findobjcatextreme"
`raw` → "find the largest apple in the bedroom"
`parameters` → (`obj_property` → "largest", `object` → "apple", `room` → "bedroom")
`incomplete` → False

If the sentence had instead been "find the largest *fruit* in the bedroom" (a category instead of an object) then there would be a key-value pair of `category` →"fruit" instead of `object` →"apple". A contract exists between the planning and speech module detailing what parameter names a given frame can have, and how many.

It could also be that there was an incomplete parse. Perhaps the phrase "find the largest apple in the bed" was heard, in which case the message would look like

`frame` → "findobjcatextreme"
`raw` → "find the largest apple in the bed"
`parameters` → (`obj_property` → "largest", `object` → "apple")
`incomplete` → True
`unexpected` → (`room` → "bed")

Sometimes a task will involve additional dialogue. For example, one of the tasks is to go to a room and ask a person their name. In this scenario, when Pepper enters the room, the frameset is changed to "name" so that after Pepper asks for a person's name, it can process their response. Another example is when Pepper is asked to manipulate an object. Since Pepper does not have this functionality, the decision was made to cause it to ask a person for help. Part of this interaction involves asking yes/no questions of a human, for which Pepper switches to the "yesno" frame.

4.2.3 Enabling GPSR through TAIG

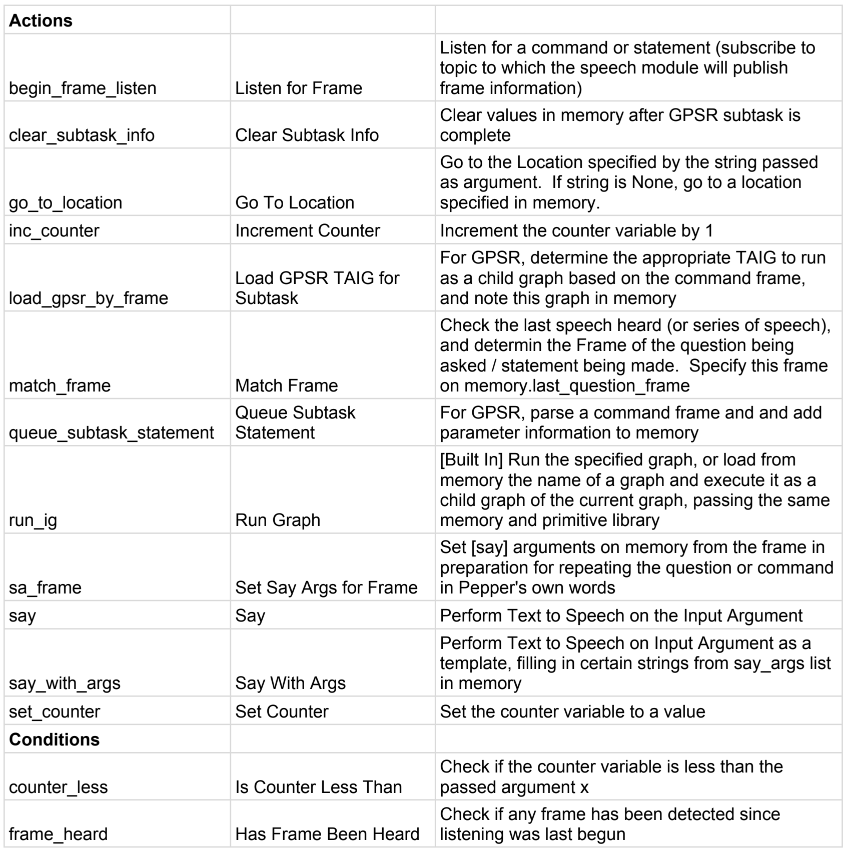
GPSR involves a complex series of actions. The possible requests that could be made of Pepper are large and involved but finite. Creating a defined task plan ahead of time, such as with a TAIG, is theoretically feasible. Certainly, it would be desirable to be able to take advantage of the specifics of the possible tasks that are known. However, the degree of variability is still such that creating the whole task as a single graph would be tedious to implement and very brittle to execute and maintain or modify. If that were our only option, it would be inferior to simply writing actual code for the task.

A more suitable approach would be to create specific plans ahead of time for executing particular components of the larger test. We can use nested graphs for this purpose.

GPSR involves listening to and executing three robot commands. We write each possible command as a graph. The GPSR test itself is also a graph (see listing 4.2 for the graph description and listing 4.1 for the legend for interpreting graph listings in this document). The GPSR-test graph involves listening for the command, and then, after parsing it, dynamically loading the appropriate graph for the specific subtask specified. It does this three times.

Another aspect of the approach is the halt condition each subtask graph has. During execution of a subtask, Pepper displays a “Stop Subtask” button on it’s tablet. When this button is pressed, the subtask stops execution immediately. The purpose of this is to account for situations in which Pepper gets stuck or is otherwise unable to continue with a particular subtask. We can use the RoboCup@Home Continue rule, and press the button. Control flow returns to the parent graph and Pepper will return to the starting position and wait for the next command (if any remain). Although continuing at this point would make us unable to achieve points for that particular command, it leaves open the possibility to achieve points for executing additional commands during the test.

The graph for the GPSR parent task in is shown in listing 4.2 and a corresponding subset of primitives from the Primitive Library is shown in Figure 4.4 (the columns are ID, Name, and Description, respectively).



Actions		
begin_frame_listen	Listen for Frame	Listen for a command or statement (subscribe to topic to which the speech module will publish frame information)
clear_subtask_info	Clear Subtask Info	Clear values in memory after GPSR subtask is complete
go_to_location	Go To Location	Go to the Location specified by the string passed as argument. If string is None, go to a location specified in memory.
inc_counter	Increment Counter	Increment the counter variable by 1
load_gpsr_by_frame	Load GPSR TAIG for Subtask	For GPSR, determine the appropriate TAIG to run as a child graph based on the command frame, and note this graph in memory
match_frame	Match Frame	Check the last speech heard (or series of speech), and determin the Frame of the question being asked / statement being made. Specify this frame on memory.last_question_frame
queue_subtask_statement	Queue Subtask Statement	For GPSR, parse a command frame and and add parameter information to memory
run_ig	Run Graph	[Built In] Run the specified graph, or load from memory the name of a graph and execute it as a child graph of the current graph, passing the same memory and primitive library
sa_frame	Set Say Args for Frame	Set [say] arguments on memory from the frame in preparation for repeating the question or command in Pepper's own words
say	Say	Perform Text to Speech on the Input Argument
say_with_args	Say With Args	Perform Text to Speech on Input Argument as a template, filling in certain strings from say_args list in memory
set_counter	Set Counter	Set the counter variable to a value
Conditions		
counter_less	Is Counter Less Than	Check if the counter variable is less than the passed argument x
frame_heard	Has Frame Been Heard	Check if any frame has been detected since listening was last begun

Figure 4.4: Primitives used in GPSR parent TAIG.

Listing 4.1: Legend for Graph Listings

```
Key:
NODE TYPE (NEGATION) primitive_id [argument list]
```

Listing 4.2: TAIG for GPSR

```
START GRAPH
say      ``Begin General Purpose Service Robot''
go_to_location  ``designated_start''
say ``Hello.  Today I am a General Purpose Service Robot.''
set_counter      [0]
WHILE counter_less      [3]
        say ``Please give me a task to perform, and when you tell
           ↪  it to me, please do not pause in the middle of
           ↪ your sentence''
        begin_frame_listen      ``gpsr_category_2''
        WHILE NOT frame_heard
        END LOOP
        match_frame
        sa_frame
        say_with_args    ``The task (task %d) that I am to perform
           ↪  is %s.''
        queue_subtask_statement ``planner_config/planner/subtask
           ↪ ''
        load_gpsr_by_frame
        run_ig
        clear_subtask_statement ``planner_config/planner/subtask
           ↪ ''
        inc_counter
        go_to_location ``designated_start''
END LOOP
say ``GPSR has been completed''
END GRAPH
```

4.2.4 Description of a Specific GSPR Command Execution: “Deliver Object From Location” Task

This section describes the partial execution of GPSR by exploring how it handles an example first command.

In the beginning of the GPSR task, the robot goes to designated start, says “Please give me a task to perform...” and listens for a command (as described in

Listing 4.2 in Section 4.2). While a frame is not heard, the robot is kept in the while loop listening for a frame.

Let's say the command the operator gives it is **"go to the dining table, find the crackers, and bring it to Charlie at the sink."**

First, this will be parsed by the speech module, which will match it to the "deliverobjfrom" frame ("Deliver Object From" frame). The "Deliver Object From" frame refers to the category of command that could be described as "Go to a location, find an object and deliver it to the person at another place." It has four required parameters (placement, object, name, and beacon). Thus, a ROS message published by the speech node after processing the above sentence would be:

`frame` → "deliverobjfrom"
`raw` → "go to the dining table find the crackers and bring it to Charlie at the sink"
`parameters` → (`name` → "Charlie", `beacon` → "sink", `object` → "crackers", `placement` → "dining table")
`incomplete` → False

When this hits the topic, the controller consumes it and the **frame_heard** primitive condition becomes true, exiting the waiting loop. The robot repeats to the human the task it is to perform (e.g. processing the ROS msg). Then, it adds to the Memory Object the information from the parameters, and sets in the memory which graph to run as a child graph (in this case, "deliverobjfrom.ig"). Then it runs that graph as a nested child graph.

The Primitive Library and the Memory Object are passed to this child graph during execution, so that it can have access to the information. The details of this graph are described in listing 4.3. The primitives used in this subtask can be found in Figure 4.5 (the columns are ID, Name, and Description, respectively).

The robot announces it's intention to find an object in the room, in this case saying, "I am going to find crackers at the dining table and deliver it to Charlie at the sink." Then it will proceed to the dining table using the *go_to_location* primitive, accessing the stored location of "dining table" and going to the coordinates to which it corresponds on the map. Once at this location, the *search_for_catobj* primitives causes it to retrieve the object "crackers" from memory and search for it in the vicinity. If it cannot find it, it goes back to the operator and reports such.

If it is able to find the crackers, then it must somehow bring them back to the operator. This was an important challenge for our team, as Pepper does not have great manipulation capabilities. Due to these limitations, the team made the decision to not attempt to have Pepper try to carry anything. Instead, Pepper will begin an intelligent interaction with a human to ask for assistance.

First, it will look for a human. If it finds one, it will call out to them, identifying them by shirt color, and ask them for help. It bribes the human with candy in return for assistance. Then, it enters into dialogue with the human, asking the question in yes/no format and waiting for a response. When the response is heard, the robot proceeds accordingly. If the response is negative, the robot returns to the operator, and explains the situation. If the response is affirmative, the robot tells the human to pick up the object and follow it, at which point it leads the way back to the operator. In either case, information about the human is stored in memory so that Pepper can talk about it to the operator.

In this manner, Pepper completes the task.

The TAIG for this subtask begins on the next page.

Listing 4.3: The complete TAIG for "Deliver Object From" ("deliverobjfrom") GPSR Subtask.

```
START GRAPH
queue_subtask_statement  ``planner_config/planner/subtask''
set_say_args_deliver_obj_from
say_with_args   ``I am going to find %s at the %s and deliver it
   ↪ to %s at the %s.''
go_to_location  None
say_with_args   ``I am looking for %s. I am at %s now. I will
   ↪ deliver the object to %s at the %s later.''
search_for_catobj       None
IF catobj_found
        say_with_args   ``I found %s at the %s. I need to deliver
           ↪ it to %s at the %s.''
        approach_obj
        point
        say     ``It doesn't look like something that I can pick
           ↪ up. I am going to look for someone to help me.''
        put_down_arm
        init_head
        find_person_in_place
        IF person_found
                set_helper_info
                say_with_args   ``Excuse me person wearing a %s
                   ↪ shirt, or anyone in this room. Would you
                   ↪ help me pick up the %s here at the %s and
                   ↪ deliver it to %s at the %s?  If you do this
                   ↪  for me, my team's humans will give you
                   ↪ candy later. Please answer yes or no.''
                begin_qframe_listen     ``yesno''
                WHILE NOT helper_response_heard
                END LOOP
                IF positive_response
                        say     ``Ok.  Please pick up the object
                           ↪ and follow me.''
                        go_to_location  None
                        say_with_args   ``The person wearing a %s
                           ↪  shirt helped me bring the %s at
                           ↪ the %s here. I am now looking for %
                           ↪ s. I am at the %s now.''
                        find_person_in_place
                        IF person_found
                                say ``I see a person here. Excuse
```

```
                                    ↪  me, the person wearing a %
                                    ↪ s helped me bring the %s at
                                    ↪  the %s over.  You must be
                                    ↪ %s at the %s. Please take
                                    ↪ the object from my helper
                                    ↪ .''
                                say ``Thank you for helping me
                                    ↪ deliver the object.  The
                                    ↪ task is done.  I will
                                    ↪ return''
                                go_to_location  ``
                                    ↪ designated_start''
                                say_with_args ``The person
                                    ↪ wearing a %s helped me
                                    ↪ bring the %s at the %s to %
                                    ↪ s at the %s.  The task is
                                    ↪ done''
                            ELSE
                                say ``It seems like the person I
                                    ↪ am looking for is not here.
                                    ↪  Sorry, please leave the
                                    ↪ object here. I will return
                                    ↪ to my operator.''
                                go_to_location          ``
                                    ↪ Designated_start''
                                say_with_args   ``The person
                                    ↪ wearing a %s shirt helped
                                    ↪ me bring the %s at the %s,
                                    ↪ but I couldn't find %s at
                                    ↪ the %s. So we left the
                                    ↪ object there. The task is
                                    ↪ done.''
                            END IF
                        ELSE
                            say ``Ok.  Thanks for letting me know.''
                            go_to_location  ``designated_start''
                            say_with_args   ``The person wearing a %s
                                ↪  shirt would not help me bring the
                                ↪ %s at the %s to %s at the %s,
                                ↪ although I did find it. Sorry.''
                        END IF
                    ELSE
                        say ``It seems that no one is in here.  I will go
```

```
                  ↪  back.''
              go_to_location   ``designated_start''
              say_with_args    ``I found the %s at the %s, but I
                  ↪  couldn't carry it and I couldn't find
                  ↪ anyone to help me bring it to %s at the %s
                  ↪ .''
      END IF
ELSE
      say      ``I cannot find the object here.  I will go back
          ↪ .''
      go_to_location   ``designated_start''
      say_with_args    ``I could not find the %s at the %s to
          ↪ deliver to %s at the %s.  Sorry.''
END IF
END GRAPH
HALT CONDITION: button_is_pressed        ``Stop Subtask''
```

What if Pepper gets stuck? With such a complex task, this is not an impossibility. Although the developers attempted to account for most scenarios, there is a chance it could happen. In the current graph, if Pepper successfully finds the objects and sees a human near enough to talk to but the human just stands there and does not respond to Pepper's interrogation, Pepper will wait indefinitely for the human to respond. Of course, knowing of this possibility allows us to account for it in the code, but ultimately there is always the chance of the unexpected. In this situation, an operator can use the specified halt condition and press the "Stop Subtask" button. Control will return to the parent GPSR graph and the robot will have an opportunity to earn more points before time runs out.

Please find a list of all primitives used in GPSR in Appendix F and some additional subtask graphs for GPSR in Appendix E.

Without the advances in TAIG, it would not have been feasible to use instruction graph for GPSR.

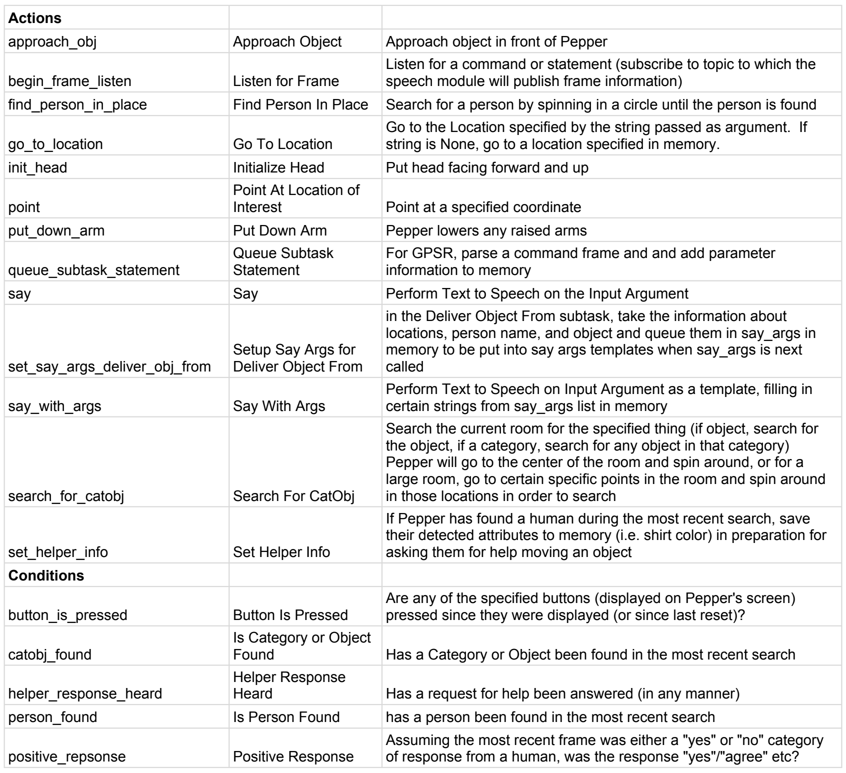
With our improvements, the use of TAIG is not only possible but advantageous. TAIG enables us to dynamically load and run subtasks using graphs-as-primitives. We use halt conditions as an emergency control flow measure that does not require stopping the entire operation of the robot.

4.3 Further Use of TAIG

The TAIG software was used during the 2018 RoboCub@Home competition and was also used in the CORAL lab for Pepper and Baxter to complete tasks.

Additionally, it was used in Human-Robot Interaction experiments conducted on

Figure 4.5: Primitives used in “Deliver Object From” TAIG.



Actions		
approach_obj	Approach Object	Approach object in front of Pepper
begin_frame_listen	Listen for Frame	Listen for a command or statement (subscribe to topic to which the speech module will publish frame information)
find_person_in_place	Find Person In Place	Search for a person by spinning in a circle until the person is found
go_to_location	Go To Location	Go to the Location specified by the string passed as argument. If string is None, go to a location specified in memory.
init_head	Initialize Head	Put head facing forward and up
point	Point At Location of Interest	Point at a specified coordinate
put_down_arm	Put Down Arm	Pepper lowers any raised arms
queue_subtask_statement	Queue Subtask Statement	For GPSR, parse a command frame and and add parameter information to memory
say	Say	Perform Text to Speech on the Input Argument
set_say_args_deliver_obj_from	Setup Say Args for Deliver Object From	in the Deliver Object From subtask, take the information about locations, person name, and object and queue them in say_args in memory to be put into say args templates when say_args is next called
say_with_args	Say With Args	Perform Text to Speech on Input Argument as a template, filling in certain strings from say_args list in memory
search_for_catobj	Search For CatObj	Search the current room for the specified thing (if object, search for the object, if a category, search for any object in that category) Pepper will go to the center of the room and spin around, or for a large room, go to certain specific points in the room and spin around in those locations in order to search
set_helper_info	Set Helper Info	If Pepper has found a human during the most recent search, save their detected attributes to memory (i.e. shirt color) in preparation for asking them for help moving an object
Conditions		
button_is_pressed	Button Is Pressed	Are any of the specified buttons (displayed on Pepper's screen) pressed since they were displayed (or since last reset)?
catobj_found	Is Category or Object Found	Has a Category or Object been found in the most recent search
helper_response_heard	Helper Response Heard	Has a request for help been answered (in any manner)
person_found	Is Person Found	has a person been found in the most recent search
positive_repsonse	Positive Response	Assuming the most recent frame was either a "yes" or "no" category of response from a human, was the response "yes"/"agree" etc?

the CMU campus that involved analyzing a human's response to Pepper's behaviors during a competitive game [78, 79].

4.4 Summary

This chapter provided real-world demonstrations of the feasibility and usefulness of TAIG. All demonstrations were successful. TAIGs on the Pepper robot showed how the use of innovations such as the Memory Object and halt conditions enabled more complex tasks to be represented and executed than could be done with an IG.

Implementing GPSR would have been tedious to create as a graph if nesting were not an option. With nesting and halt conditions, it is not only feasible but beneficial.

Another task (the memory game) was transferred between two robots (the Pepper and Baxter), where the exact same policy representation was used to execute the task on both systems. Implementing the memory game was possible to do in a transferable manner due to the introduction of memory and the primitive library. This shows how TAIG enables task transfer.

Chapter 5

Interactive Transferable Augmented Instruction Graph

Chapters 3 and 4 focused on the ability to develop, modify, and transfer a task plan between executions.

Chapter 5 builds on this foundation and discusses modifications made to the TAIG framework that allow a human to interact with a computer/robot agent to build a TAIG ***interactively*** during runtime. This contributes towards a future that includes lay people using natural verbal interaction with robots as a means of creating task plans.

Through the work presented in this chapter, a human will be able to have a conversation with this agent according to a set grammar. Through this grammar, the human will be able to command execution of command primitives or existing task graphs as well as build up new task graphs that can be executed.

This marks another improvement over the simple IG, in the following ways. Although similar interactive work was done regarding the simple IG [58], Interactive-TAIG is included in the TAIG library and is a framework that is fully customizable by the end user. New primitives can be added in to the framework to be recognized, and even the grammar itself can be adjusted. Additionally, interactive-TAIG supports the creation of nested graphs during the interactive process.

Section 5.1 introduces a formalization of the additions to TAIG that enable interactivity. It involves updating the Primitive Tuple and introducing the Interactive Manager. Section 5.2 demonstrates creating a TAIG via this interactive process.

Interactive-TAIG is an extension to TAIG that allows for interactivity in a situation where the use of a TAIG might be desired.

5.1 Methods for Realizing Runtime Interactivity

There are two main components to the extension. First, the primitive tuple is expanded with additional elements. Primarily this is to allow the robot and human a means of referring to a primitive.

The second main component is the "Interactive Manager" (IM), which serves as the agent with which the human interacts.

5.1.1 The Updated Primitive Tuple

In the pursuit of interactivity, we do not only care what a primitive does, but also how a sentence can refer to it, both as a request and as a description. Thus `PT` shown in Equation 3.5 is expanded and updated as per Equation 5.1. This may entail some additional up-front development cost per primitive.

The additional elements enable the following functionality:

- Specifying how the human (or non-IM agent) refers to this primitive
- Specifying how a string of a text describes the parameters of this primitive
- Specifying how the IM agent refers to this primitive, both in general and with a specific parameterization

All the new elements are optional, but should be filled in if the primitive is to be used in an interactive manner. It is explicitly not required to include them and their presence or lack thereof has no effect on the technical operation of an instruction graph.

$$\begin{aligned}
\mathrm{PT}_i =< \mathrm{pid}, t, f, \mathrm{name}, \mathrm{description}, \\
\mathrm{matchFn}, \mathrm{argparseFn}, \mathrm{paramDescFn} > \\
t \in \{\text{Action,Condition}\} \\
\mathrm{matchFn} : (\texttt{string} \rightarrow \{\mathrm{True}, \mathrm{False}\}) \\
\mathrm{argparseFn} : (\texttt{string} \rightarrow \texttt{list[string]}) \\
\mathrm{paramDescFn} : (\texttt{list[string]} \rightarrow \texttt{string}) \\
\text{s.t. } \mathrm{PT}_i[\mathrm{pid}] \neq \mathrm{PT}_j[\mathrm{pid}] \quad \forall i \neq j \quad \text{where } \mathrm{PT}_i, \mathrm{PT}_j \in \mathrm{PL}
\end{aligned} \tag{5.1}$$

where the new elements `name`, `description`, `matchFn`, `argparseFn`, and `paramDescFn` are described below. As in Equation 3.6, `pid` is the primitive identifier, t is the type, and f is the literal function. To use the interactive agent, at least `matchFn` (and `argparseFn` if you want any parameters to be parsed) must be filled in.

Human-Readable Name and Description

`name` is a human-readable simple-name for the primitive and `description` is a human-readable description meant to give information about what the function does, any arguments it might take, what should be supplied, and other useful long-form information.

Human Understandability Even Without Interactivity These two attributes would be useful to fill in even if a developer was handcrafting tasks and not using the interactive module. It would benefit a human developer, too.

In a real-world complex robotics system, there could be a large number of primitive actions and conditions. Moreover, it may not be clear from the function name what the entire purpose of a primitive is, or what arguments it requires. It would be inefficient and defeat the purpose of using the TAIG paradigm if it required a human to read and becomes familiar with the code of each primitive in order to decide if they should tell a robot to include it in an instruction graph.

In this manner, the benefit the `name` and `descripition` elements still provide is understanding the purpose and behavior of a primitive without looking through the code. A human user creating an IG can use this to quickly create an IG even from a Primitive Library with which they are previously unfamiliar. An additional benefit is that when an IG is automatically generated by a robot or other machine process (as in [47]), a human could readily understand what the new IG does by inspecting these attributes.

Match Function or Regex

The element `matchFn` is a function that checks an input text string (from a human or other agent) and determines if it refers to the primitive $\mathtt{PT}_i$ (returning a boolean `True` if so). In the instruction_graph library, this element can be a function or it can be a regular expression (regex) [53]. If it is a regex, it is transformed into a function that returns `True` if the regex matches the input string.

This must be specified in order for the IM to recognize a string of written text as referring to this primitive.

Argument Parsing Function or Regex

The element `argparseFn` is a function that takes as input a string (previously confirmed to refer to the primitive $\mathtt{PT}_i$) and returns a list of strings that are the parameter values for the specific parameterization of this primitive.

In the instruction_graph library, `argparseFn` can be a function or it can be a regex. If it is a regex with capturing groups, it is transformed into a function that returns a list of strings, where each string is the contents of a captured group.

If the primitive has no parameters, this value can be ignored. Otherwise it should be specified in order for the IM to parse a string of written text into a specific parameterization.

Parameterized Description Function or Regex

The element `paramDesFn` is a function that maps from a list of strings (representing parameters) to a string. The output string is a natural language description of this primitive parameterized with the input parameters. This is used when the IM agent wishes to describe an instance of a primitive. The output should be human-understandable.

If `paramDesFn` is not supplied, the IM agent will fall back to `description` instead. If neither of these values is supplied, then the IM has no way to describe the primitive in a human-intelligible manner.

Example

Consider the following mini-example. Imagine that there is a library with the following two primitives defined:

$$\begin{aligned}
\mathrm{PT}_1 = &< 1, \mathrm{Action}, f_1, \text{“Rotate”}, \text{”Rotate in place.”}, \\
&\quad \mathrm{matchFn} : \texttt{r``rotate .* (degrees|radians)''} \\
&\quad \mathrm{argparseFn} : \texttt{r``rotate (.*) (degrees|radians)''} \\
&\quad \mathrm{paramDescFn} : \text{“rotate \%s \%s”} > \\
\mathrm{PT}_2 = &< 2, \mathrm{Action}, f_2, \text{“Move”}, \text{“Move in a forward/back.”}, \\
&\quad \mathrm{matchFn} : \texttt{r``move (forward|back) .* (foot|feet|meters)''} \\
&\quad \mathrm{argparseFn} : \texttt{r``move (forward|back) (.*) (foot|feet|meters)''} \\
&\quad \mathrm{paramDescFn} : \text{“Move in the \%s direction by \%s \%s”} >
\end{aligned} \tag{5.2}$$

The `r` prepending the string indicates a regular expression to be tested.

Consider that a phrase “Please move forward three meters.” is heard by the agent. (The specific processing of this will be covered in the next section.) To determine whether this corresponds to any primitives, the `matchFn` attribute is checked, and the regular expression contained within is run against the phrase. In this case, it would match $\texttt{PT}_2$ but not $\texttt{PT}_1$.

Now, consider that $\texttt{PT}_2$ is matched, and it is now required to determine how to parameterize the node tuple that will refer to this primitive tuple. This is done by accessing the `argparseFn` attribute of $\texttt{PT}_2$. In this example, it is a regular expression with capturing groups. Three arguments will thus be captured, and stored as parameters. (Thus it can be inferred that it is valid for f_2 to take three parameters. It is important that the output of argparseFn and input of f are compatible.) In this case, the three parameters would be "forward", "three", and "meters".

If the agent needs to describe this node in the graph, it will use `paramDescFn` to describe it as "Move in the forward direction by three meters."

The example here uses the regex and string conventions, with the behavior described. Creating explicit functions to match, parse, and describe is also permitted.

5.1.2 The Interactive Manager

The goal of the Interactive Manager is to allow command and instruction of the robot or agent in the execution and construction of TAIGs via natural language. By default, there is a text-based interface. It is possible to connect it to an external speech system to allow for verbal communication as well.

The grammar is a combination of library-defined structures and primitive-defined structures.

On instantiation, the Interactive Manager refers to a specific Primitive Library and (optionally) Memory Object. This controls what primitives are available.

Additionally, there is a default set of a "Builder Phrases" which can be modified as necessary. This controls the grammar (and so the grammar is also somewhat configurable). There is a set of grammar components that are always present, but the specific regex or string corresponding to it can be changed. For example, the default for an affirmative response is "yes" but this could be changed to "yup," "righty-o," or any other phrase desired. Another example is that the default phrase for a human to tell the IM when learning a task is done is any pattern satisfying the regex

```
done (?:learning|teaching)
```

but this could be changed to any regex. The details of the Builder Phrases are shown in Appendix B and an example interaction is shown in Section 5.2.

Interaction with the IM is achieved through repeatedly calling the **parse_input_text** method. It takes one argument, a string of text from the human or other agent, and returns a string representing the response. This simple interface means that the IM could be hooked into a variety of other systems.

The IM agent can be thought of as a finite state machine. The IM agent exists in one of four different states:

- WAITING: No task is currently being learned. Agent can be told to i) execute a primitive, ii) execute a graph, or iii) begin learning a graph. (A newly instantiated IM begins in this state.)
- CONFIRM_LEARN: The IM agent believes that it has just been asked to learn

a new task graph, and will listen for confirmation.

- LEARN_WAITING: The IM agent is currently learning a new task graph. Agent can be told to add a new primitive or be told that the graph is complete.
- CONFIRM_ADD: The IM agent believes that it has just been asked to add a primitive to a task graph, and will listen for confirmation.

The IM maintains the following attributes:

- state $\in \{\text{WAITING}, \text{CONFIRM\_LEARN}, \text{LEARN\_WAITING}, \text{CONFIRM\_ADD}\}$
- ig_dir: the directory where existing graph files are to be found and new graphs are to be stored
- p: Builder Phrases
- ig_name: `None` during WAITING, otherwise holds the name of the graph currently being built
- PT_Q: a fully parameterized Primitive Tuple queued to be added. (Will be empty in all states aside from CONFIRM_ADD.)

When the **parse_input_text** method is called, the behavior will depend on the state. In each of the four states, the raw input text is pattern-matched against a series of Builder Phrases, in the order noted in the figure. When a match occurs, the relevant action and/or state change occurs. The agent will respond appropriately, asking for confirmation in the case of a request, or stating what action it has taken (whether execution of a task/primitive, construction of a graph, or other action). If no match occurs, the agent responds with this fact, asking for another response. The state machine is described in Figure 5.1.

The orange shapes represent the four states. Each branch from each state is numbered. Each branch represents a path to follow *if* a particular pattern is matched, checked in the order of number increasing from 1 (the largest number off of each circle represents the default case if no previous matches occur). Each branch has text indicating the nature of the patterns (patterns themselves are configured and stored in Builder Phrases). A pentagon is a conditional check and a blue box represents the procedure that the agent follows (low level procedures may not be included).

For example, consider an IM in the WAITING state that receives input text "raise arm". First, the IM will check whether it is being asked to learn a new graph. It is not. Second, it checks whether it is asked to run a graph. It is not. Next, it will check whether there is a primitive in the library that matches "raise arm". If there is, it will be executed at this time. If the input text was instead "run raise arm", this matches the Phrase for running a graph, and so it would attempt to run a task graph with the name "raise arm", regardless of whether or not there was a primitive with that designation as well. If the input text is "I will teach you to raise arm" the agent would assign "raise arm" to `ig_name`, move to state CONFIRM_LEARN, and ask for confirmation to learn this new graph. These general matching regular expressions are

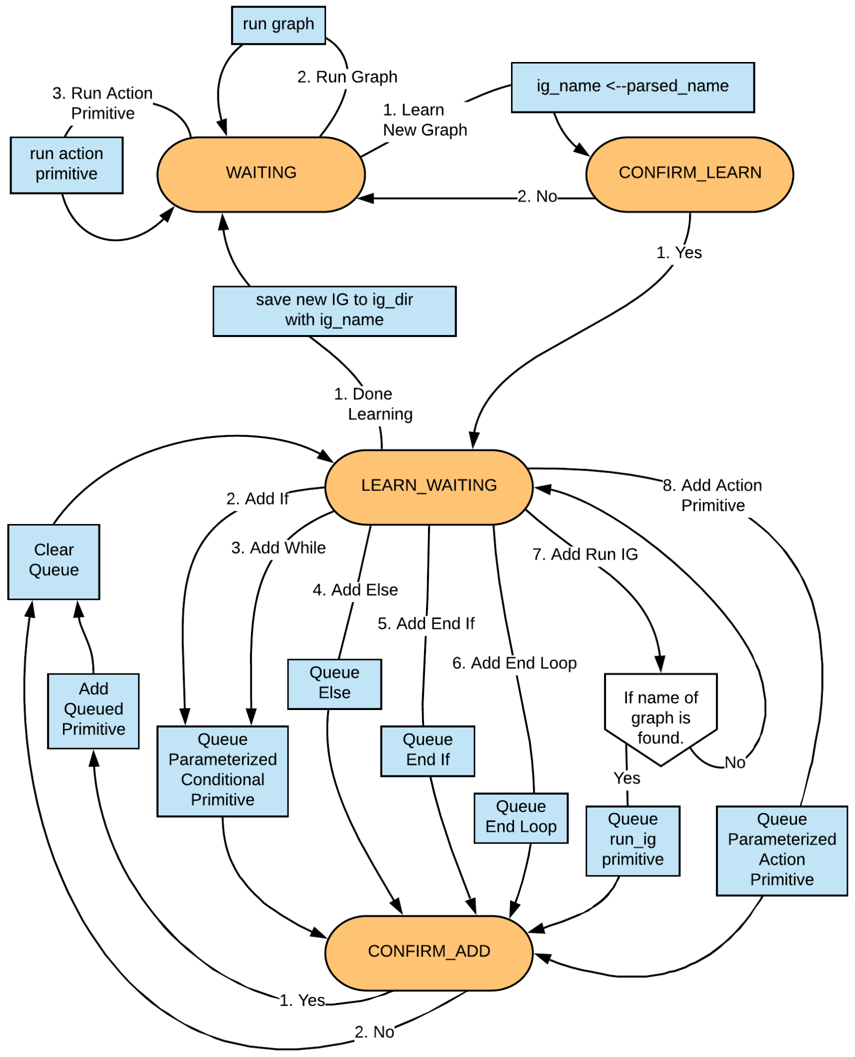


Figure 5.1: A finite state machine representing the Interactive Manager's behavior

in the Builder Phrases and can be modified before execution if desired.

The extra elements in the Primitive Tuple discussed in Equation 5.1 in Section 5.1.1 enable some of this functionality. The details of these cases are discussed in the following subsections.

Executing a Task Graph

In the WAITING state, if the "run task graph" pattern is matched (which by default settings would be "run [task-graph-name]"), the `ig_dir` directory is searched for a graph with this name. Both pre-existing and newly created task graph files will be stored here. If one is found, it is loaded and executed immediately. If no such graph is found, the agent says it cannot find a graph by this name.

One limitation discovered in the course of implementing this example is that in the case of audio conversation, names that are spelled differently but sound the same are an issue. Of course, when communicating via typed text this is not a problem.

Executing an Action Primitive

In the WAITING state, if neither the "learn new graph" nor "run graph" patterns are matched, the agent assumes it is being instructed to execute a single primitive. It iterates through the entire primitive library, executing each `matchFn` until a match occurs. (If no match occurs, the agent responds with this information.) Then, if there is a match, the `argparseFn` is used to parameterize the instance of the Primitive Tuple, which is then immediately executed with those parameters. The `paramDesFn` (or `description`) is used to describe back to the user what the agent is now executing.

Adding an Action Primitive to a Task Graph

In the LEARN_WAITING state, if none of the patterns 1 through 7 are matched, branch 8 is the "Add Action Primitive". The agent will attempt to match input text to a Primitive Tuple using `matchFn`, and if a there is a match, parameterize it using `argparseFn`. This information is stored in $\texttt{PT}_Q$. The `paramDesFn` is used for the agent to ask confirmation of the user as to whether what the agent believes is the requested node to create is actually so, and the `state` is set to CONFIRM_ADD.

In the state CONFIRM_ADD, if the user gives an affirmative text input, the Primitive Tuple stored in $\texttt{PT}_Q$ becomes added to the task graph. If the user gives a negative text input, $\texttt{PT}_Q$ is simply cleared. Either way, the state returns to LEARN_WAITING.

Adding a Conditional Primitive to a Task Graph

In the LEARN_WAITING state, if the "Add If" or "Add While" (branch 2 or 3) is recognized, a conditional Primitive Tuple is retrieved, parameterized, and added to the queue in much the same way as for an action Primitive Tuple. The confirmation is the same as well.

The difference is that there will be a keyword such as "If" or "While" that tells the agent that the text following will specify a Conditional type of Primitive Tuple and not an Action.

Adding a Nested Graph (run_ig) as a Node to a Task Graph

When in the LEARN_WAITING state, a Run IG node can be added to the task graph (nesting an existing graph inside a new graph). The syntax for this is exactly the same as the syntax for requesting to execute a graph in the WAITING state. When the command is received in the LEARN_WAITING state, however, instead of executing the graph, the agent will ask for confirmation and then add it as a node in the graph currently under construction.

5.2 Learning to Search and Deliver a Message

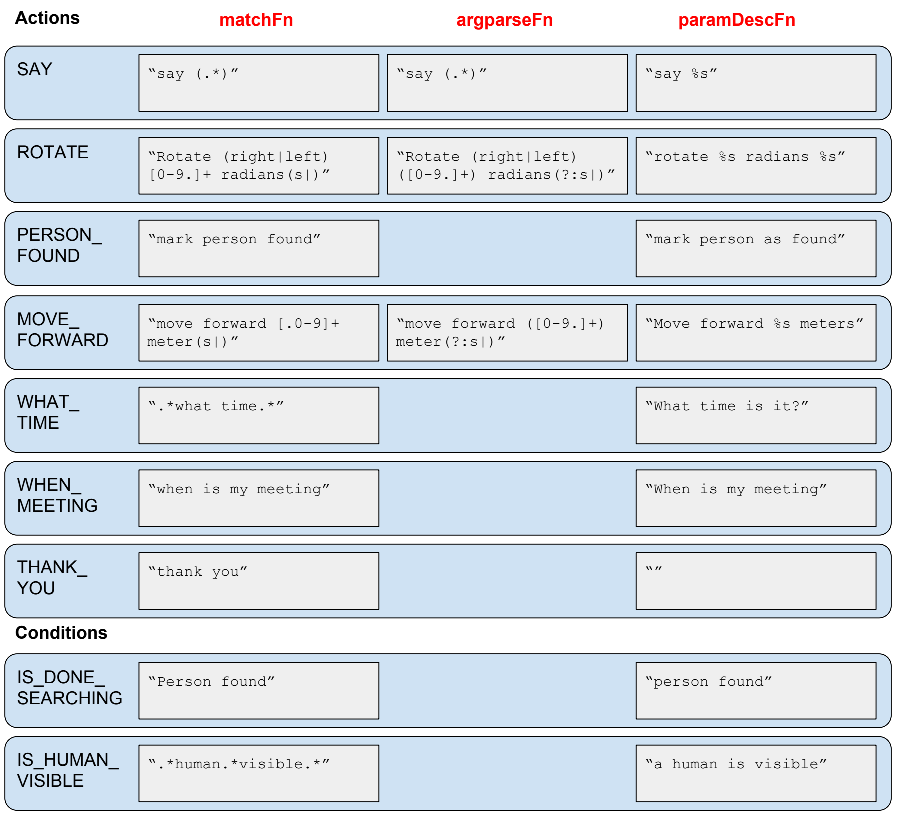
We demonstrate the Interactive TAIG capability on the Pepper robot with a search-and-deliver-message task.

The imagined scenario is that a human operator wants to teach Pepper to find them and tell them when their next meeting is. To make the scenario simplistic, 'searching' is simply rotating in place until a person is found.

There are nine primitives used in this demonstration, two conditional and seven action. Find them described in Figure 5.2 (with interactive-enabling attributes noted).[1] ((The `paramDescFn` (or **parsed_description**) attribute is a function but in the figure it is just noted as a string that conveys the result. For example, the actual value for the ROTATE primitive is
`lambda args: "rotate %s radians %s" % (args[0], args[1])`,
abbreviated as the value shown in the figure.)

During the demonstration, according to the dialogue shown later, two graphs are created. The first graph is a "spin search" graph, where Pepper learns to spin around until it sees a human. When it sees a human it stops and the task completes. This is

[1]Find the full code file at `https://github.com/AMR-/CMU-Structured-Representations-For-Robots/blob/master/Interactive_TAIG/pepper_ig/ITAIGDemoPrimitiveLibrary.py` If you view the code, note also the following difference between the names of the attributes in the code and in this thesis: `matchFn` becomes **match_re_or_fn**, `argparseFn` becomes **argparse_re_or_fn**, and `paramDescFn` becomes **parsed_description**.



Actions	matchFn	argparseFn	paramDescFn
SAY	“say (.*)”	“say (.*)”	“say %s”
ROTATE	“Rotate (right|left) [0-9.]+ radians(s|)”	“Rotate (right|left) ([0-9.]+) radians(?:s|)”	“rotate %s radians %s”
PERSON_ FOUND	“mark person found”		“mark person as found”
MOVE_ FORWARD	“move forward [.0-9]+ meter(s|)”	“move forward ([0-9.]+) meter(?:s|)”	“Move forward %s meters”
WHAT_ TIME	“.*what time.*”		“What time is it?”
WHEN_ MEETING	“when is my meeting”		“When is my meeting”
THANK_ YOU	“thank you”		“”
Conditions			
IS_DONE_ SEARCHING	“Person found”		“person found”
IS_HUMAN_ VISIBLE	“.*human.*visible.*”		“a human is visible”

Figure 5.2: Description of primitives in Primitive Library used in the Search and Deliver Message task

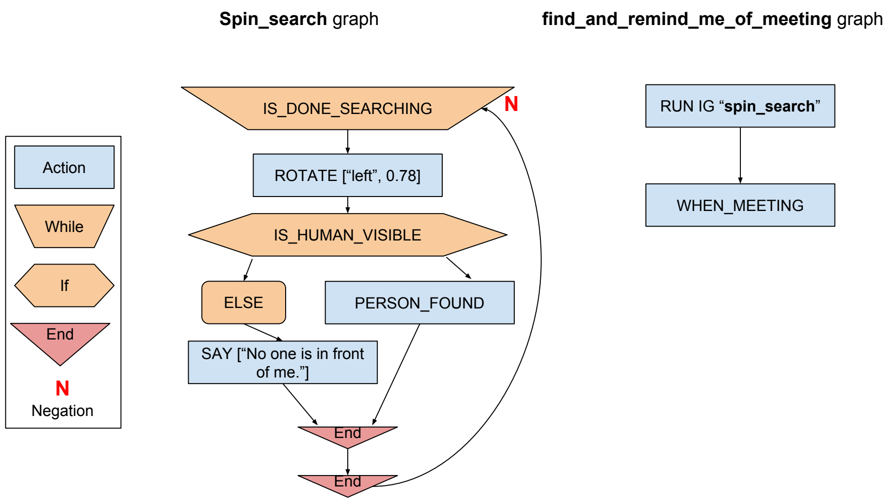


Figure 5.3: Description of the task graphs that are created during the demonstrated dialogue

used as a subtask in the "find and remind me of a meeting" graph, where Pepper will spin around until it sees a human, after which it will state when is the person's meeting. These two graphs are shown in Figure 5.3. The graphs are created entirely using the interactive process.

To use interactive TAIG with Pepper, a simple script[2] was made that uses Pepper's in-built speech recognition to translate spoken words into text which is then fed to the interactive manager. (Since voice recognition is not a part of this work, a limited vocabulary specific to this demonstration was defined for Pepper, increasing the chance that Pepper would hear our phrases correctly. The text was then fed into the interactive manager in raw form.) The text response from the interactive manager (if any), is then spoken by Pepper.

Find the actual dialogue used in the demonstration in Figure 5.4. 'H' refers to the human and 'P' refers to Pepper.

The dialogue begins with the human giving commands to Pepper, with Pepper executing the commands (telling the current time and moving forward a meter). You can see that the command matches the `matchFn` attribute of the WHAT_TIME and MOVE_FORWARD primitives.

Then, the human indicates that Pepper should learn a new graph "spin_search". The first command in this graph is a to create a while loop. Note how the command "While not person found" indicates the while loop, indicates the negation, and identifies

[2]Find example code for using TAIG interactive manager with Pepper here: https://github.com/AMR-/CMU-Structured-Representations-For-Robots/tree/master/Interactive_TAIG

H: Pepper, what time is it?
P: Current time is [time]
P: Executed 'what time is it?'
H: Move forward one meter.
P: [moves forward]
P: Executed move forward one meters
H: I will teach you to spin_search
P: I will learn to spin search?
H: Yes
P: Ok, I am ready to learn. What is first?
H: While not person found
P: I should a start a while loop with negated condition 'person found'?
H: Yes
P: Ok, what's next?
H: Rotate left 0.78 radians
P: I should rotate by radians 0.78?
H: Yes
P: Ok, what's next?
H: If human is visible
P: I should start an IF condition with condition 'if human is visible'?
H: Yes
P: Ok, what's next?
H: Mark person found
P: I should mark person as found?
H: Yes
P: Ok, what's next?
H: ELSE
P: I should add an ELSE clause to the IF condition?
H: Yes
P: Ok, what's next?
H: Say "No one is in front of me."
P: I should say 'no one is in front of me'?
H: Yes
P: Ok, what's next?
H: End if
P: I should end the IF condition?
H: Yes
P: Okay, what's next?
H: End loop
P: I should end the loop?
H: Yes
P: Ok, what's next?
H: Done Learning
P: I have learned spin_search
H: I will teach you to find me and remind me of a meeting
P: I will learn to find me and remind me of a meeting
H: Yes
P: Ok, what's first?
H: Run spin search
P: I should run the graph spin_search?
H: Yes
P: Ok, what's next?
H: State meeting time
P: I should 'when is my meeting?'
H: Yes
P: Ok, what's next?
H: Done learning
P: I have learned 'find_me_and_remind_me_of_a_meeting'
H: Run find and remind me of a meeting
(P executes task)
H: Thank you.
P: You are welcome

Figure 5.4: Full dialogue for the demonstration

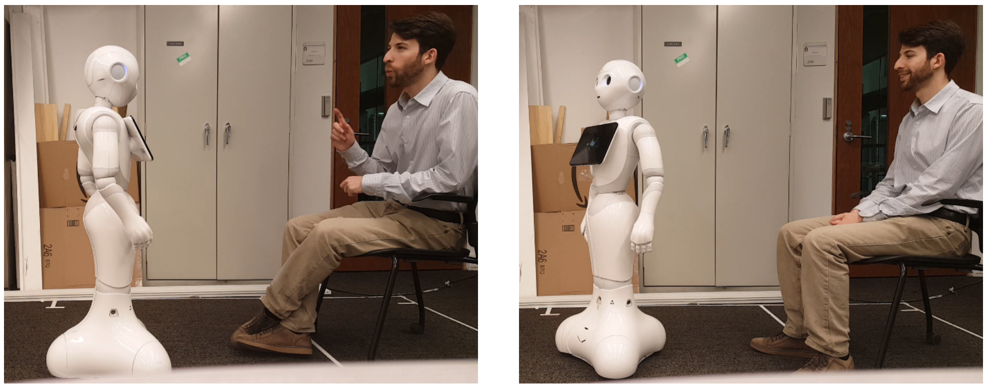


Figure 5.5: Pepper learns and executes the "find-and-remind-me-of-a-meeting" task.

the condition for the loop. The interactive manager's Builder Phrases tell it how to parse the fact that it is a while loop and how to parse the negation, and the `matchFn` attribute on the IS_DONE_SEARCHING primitive causes the match.

The second command to add to the graph is "Rotate left 0.78 radians". This adds a parameterized action primitive. Once again, the `matchFn` attribute on the ROTATE primitive causes the match. The `argparseFn` attribute allows the interactive manager to parse the two arguments from the command: "left" and "0.78". The `paramDescFn` attribute describes how Pepper will phrase the intended primitive with parameters when it requests confirmation from the human.

The graph shown in Figure 5.3 is created via this interactive process.

After the "spin_search" graph is created, the human in the demonstration creates a "find_and_remind_me_of_a_meeting" graph. This graph has as it's first action primitive a run_ig primitive that will run a nested graph named "spin_search".

Finally, after creating these graphs, the human user commands pepper to execute the "find_and_remind_me_of_a_meeting" task. Pepper then spins around until it sees the human again, and reminds them of the meeting. See a sequence of pictures depicting some moments from this process in Figure 5.5. Videos of this demonstration can be downloaded from the github video repository associated with this thesis.[3]

[3] A videos of this interaction can be downloaded from `https://github.com/AMR-/CMU-Structured-Representations-For-Robots-Videos/blob/master/ITAIG_raw.mp4`. The same video with visual annotations corresponding to the robot's state and understanding of the in-construction task graph can be downloaded from `https://github.com/AMR-/CMU-Structured-Representations-For-Robots-Videos/blob/master/ITAIG_all_annotations.mp4`.

5.3 Summary

This chapter introduced an extension to TAIG that facilitates interactive dialogues with a computer or robot agent, through which TAIGs can be created, saved, loaded, and run. This helps to make TAIG useful in the world, as it is no longer strictly necessary to use programming to create or run a TAIG. These advances are included in the open-source library. With the right groundwork (a sufficiently defined Primitive Library), even a lay-person could interact with an agent to run commands or give instruction on a wholly new task.

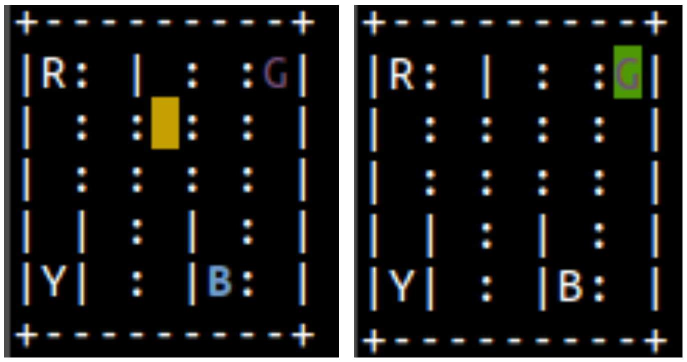
Chapter 6

Explainable Reinforcement Learning via Decision Tree Policies: Using Human-Defined Conditions

This chapter switches focus from constructing structured policies under human direction to learning structured policies with a combination of human input and autonomous learning. When autonomous learning is brought into the picture, it could be asked whether keeping policy structured matters. There are reasons for structure, one of which is that it allows for interpretability. The benefits of interpretability are further discussed in Chapter 7. A human may be able to determine and define possible actions and conditions, but not determine optimal or even a successful policy. (In other words, creating a Primitive Library, but without creating a successful graph.)

Chapter 6 begins an initial investigation into constructing policies in a decision tree format using reinforcement learning. It discusses policies that take the form of a decision tree where the conditional branches of the tree are split on features defined by a human. A human will create a set of possible conditions/features that the agent can choose to utilize, and our methods will determine which ones to use and where to place them in a tree.

The first method, described in Section 6.2, learns a tabular policy using reinforcement learning that our algorithm transforms into a decision tree. The next method, described in Section 6.3, learns a decision tree itself using a reinforcement learning algorithm. The domain used in each case, the Taxi domain, is explained in Section 6.1.



```
+---------+   +---------+
|R: | : :G|   |R: | : :G|
| : : : : |   | : : : : |
| : : : : |   | : : : : |
| | : | : |   | | : | : |
|Y| : |B: |   |Y| : |B: |
+---------+   +---------+
```

Figure 6.1: Visual representation of the taxi environment.

6.1 Taxi Environment

In our explorations with human-defined conditions, the Taxi environment "Taxi-V2" [30], from Open AI [19], is used.[1] The Taxi environment is a 5x5 grid. The taxi can drive around. Obstacles are represented by '—' and open space by ':'. There are four locations where passengers can stand to wait to be picked up (marked by the letters 'R', 'G', 'Y', 'B'). A passenger will start at one of these locations and want to go to a different one of the same four locations. The goal is to find a policy for the taxi such that wherever the taxi starts it will go to the passenger, pick up the passenger, go to the goal location, and drop off the passenger. This discrete environment has 500 total states and 6 basic actions. Two examples are shown in Figure 6.1. On left, the taxi (yellow rectangle) is at position (1,2), the passenger to be picked up is in position 3 ('B'), and the drop-off location is in position 1 ('G'). The taxi turns green when the passenger is inside it. On the right, the episode is complete, as the taxi has dropped off the passenger at their destination.

6.2 Transformation of an Existing Tabular Policy

6.2.1 Method

One method of building a decision-tree-style policy is to take an existing policy and transform it into a decision tree style policy.

The method proposed here transforms a Q-table $\mathbb{Q}$ from a tabular policy into a decision tree. The idea is to group states that are similar (with similar features and the same 'best action') into buckets based on conditions.

[1]https://gym.openai.com/envs/Taxi-v2/

In the transformation method, a human specifies a set of conditions C, where each $c \in C$ is a function mapping from a state s to a boolean B.

Given C and $\mathbb{Q}$, the process has two steps: i) checking whether C is "sufficient" to separate the policy into a decision tree, and ii) transforming the policy.

Algorithm 1: Procedure for Determining If a Set of Conditions is "Sufficient" with Respect to a Given Tabular Policy

```
1  Given set of conditions $C$ where each $(c : s \rightarrow B) \in C$    (meaning, each $c$ in $C$
     is a function taking a state $s$ as an argument and returning a boolean).
2  Given environment $E$ with set of possible states $s \in S$
3  Given a Q-table which maps each $s \in S$ to a next_action
4  -
5  $g \in$ mixed_groups $\leftarrow$ a list of sets, where each set $g \subset S$ and sets in the list are
     non-intersecting (mutually exclusive);
6  % initialize mixed_groups as a list with a single item - the set $S$ itself
7  mixed_groups $\leftarrow [\{S\}]$;
8  $D \leftarrow C$;
9  while $mixed\_groups \neq \emptyset$ and $D \neq \emptyset$ do
10     $c \leftarrow$ condition drawn from and removed from $D$;
11     split_groups $\leftarrow$ (the list of sets derived by taking each set in mixed_groups
         and separating that set into two sets – one set containing each $s$ for which
         $c(s)$ is true and one set containing each $s$ for which $c(s)$ is false);
12     Remove from split_groups all empty sets;
13     Remove from split_groups all sets where each state $s$ in the set yields the
         same next_action according to the Q-table;
14     mixed_groups $\leftarrow$ split_groups;
15 end
16 if $mixed\_groups = \emptyset$ then
17     The conditions in $C$ are sufficient
18 else
19     The conditions in $C$ are insufficient
20 end
```

The algorithm for step (i) is shown in Algorithm 1. If C is marked "sufficient", then C can be used to create a decision tree that replicates the policy of the table precisely. It is not always necessary for a set of conditions C to be marked as sufficient in order to obtain a policy that could solve a task—it may be the case that a policy that *almost* replicates the tabular policy is good enough. But this check provides a guarantee, and even if C is not sufficient, exploring what states remain in mixed_groups at the end of a sufficiency check can allow the human to intelligently determine new conditions to add to C. It might also be that a decision tree using sufficient conditions would overfit, and that using insufficient conditions is a superior

state	F1	F2	F3	action
A	0	0	0	3
B	0	1	1	4
C	1	1	1	5
D	1	1	1	5
E	0	0	1	4
F	1	0	1	5
G	0	0	0	3
H	1	1	0	3

Table 6.1: Example of sufficiency concept: states and features

choice. (This is always environment and condition dependent.)

Algorithm 2: Transforming a Q-table into a Decision Tree Policy

```
1  Given an $n_s$ x $n_a$ q-table $\mathbb{Q}$ (where $n_s$ are the number of states $s \in S$ and $n_a$
     are the number of actions $a \in A$);
2  Given an indexed, ordered set of condition functions $(c : s \rightarrow B) \in C$ such that
     $c(s)$ yields a boolean $\forall c, s$;
3  $f \leftarrow$ a $n_s$ x $n_c$ feature matrix (where $n_c$ are the number of $c \in C$), created as
     follows:
4  for $r \leftarrow 0$ to $n_s$ do
5      for $i \leftarrow 0$ to $n_c$ do
6          % give the value of $f$ at ($r$,$i$) the binary output from executing the $i^{th}$
             condition $c_i$ on the $r^{th}$ state $s_r$
7          $f[r, i] \leftarrow c_i(s_r)$
8      end
9  end
10 $L \leftarrow$ a 1 x $n_s$ label vector :
11 for $r \leftarrow 0$ to $n_s$ do
12     $L[r] = a'$ for which value of $\mathbb{Q}(s_r, a')$ is greatest;
13 end
14 Create a decision tree using features $f$ and labels $L$ using the standard ID3
     algorithm;
```

For example, given the 7 states in Table 6.1, consider the case where the human suggests features 1 and 2 compared to the case where the human suggests features 1, 2, and 3. The results of the conditions for a given state are as shown. ("F1" means "feature 1," and the "action" is action with the highest Q-value.) With only features 1 and 2, the following groupings result: ((A, E, G), (B), (C, D, H), (F)). Two of the groups have members with different "best actions," (state A has best action 3 and

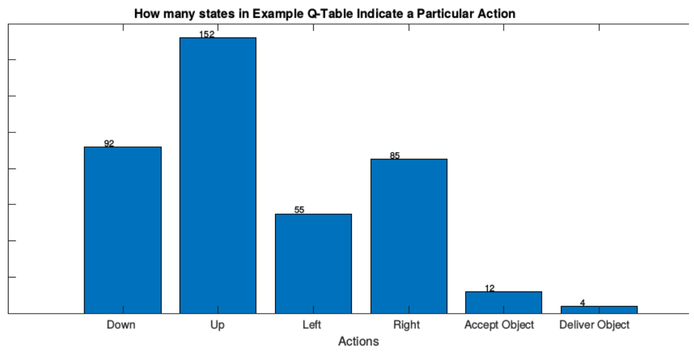


Figure 6.2: Example count of how many states indicate a particular best action for a successful policy

state E has best action 4) and so this feature set would be marked **insufficient**. It could still be chosen for use, but efficacy is not guaranteed. Alternatively, using a feature of set of features 1, 2, and 3 results in grouping ((A, G), (E), (B), (C, D), (H)). Now, the feature set is **sufficient**, because no group has multiple best outcomes. Each group could be represented as an abstract state in a leaf node of a tree.

The algorithm for step (ii), the actual transformation, is shown in Algorithm 2. We create a *condition table*, where each state is evaluated against each condition. We uses these as features for learning a decision tree. The number of classes for leaves is equal to the number of discrete actions available. We use ID3 [39] to learn the tree.

6.2.2 Exploratory Results

To demonstrate this method, the simple Taxi environment is used. The Taxi environment has 500 states, and six actions. We used standard tabular Q-learning to solve the environment.

In Figure 6.2, see an example of the distribution of how many states have a particular action as the 'best action' for an example tabular policy. It is not surprising that there are exactly 4 dropoff states (for 4 locations) or 12 pickup states (4 pickup locations x 3 goal locations for each).

In Figure 6.3 find the final list of conditions used for transforming the Taxi environment. Some of the conditions were thought of immediately (such as checks for where the passenger and goals are), while others (such as 'obstacle on the immediate right') were proposed after using Algorithm 1 to determine which sets of states were not being separated that should be. The final C used is not sufficient. There were 25 'mixed groups' remaining. Typical of this was the group (283, 183).[2] The group of size two had two states with different best actions. See renderings of each in

[2]In Taxi, states can be uniquely referenced by an integer.

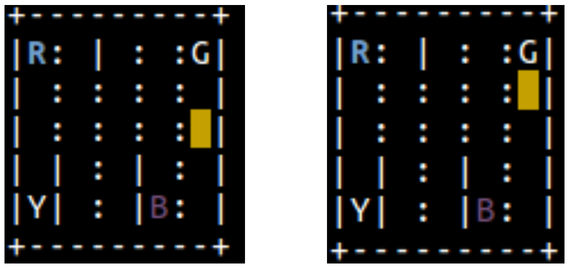
1. Taxi is at goal location
2. Destination is above the taxi
3. Destination is right of the taxi
4. Destination is below the taxi
5. Destination is left of the taxi
6. There is an obstacle between taxi and destination
7. Taxi is at passenger location
8. Passenger is in taxi
9. Passenger is below taxi
10. Passenger is above taxi
11. Passenger is right of taxi
12. Passenger is left of taxi
13. Taxi is on right edge of environment
14. Taxi is on left edge of environment
15. Taxi is on bottom edge of environment
16. Taxi is on top edge of environment
17. There is an obstacle between taxi and passenger
18. There is an obstacle on taxi's immediate left
19. There is an obstacle on taxi's immediate right

Figure 6.3: List of conditions used for taxi decision tree.

Figure 6.4. (At left is the taxi environment in state 283, and at right is the taxi environment in state 183. Both states satisfy the same set of conditions for our set of conditions.) In state 283, the taxi is at position (2,4), and the "best action" is to go Up. In state 183, the taxi is at position (1,4), and the best action is to go Left. Both of these are reasonable actions. However, is there no reason that the taxi at (2,4) also could not go Left. Ultimately, some human judgement is required in this technique to determine what set of conditions a successful policy would require.

With unlimited max depth, transformed tree for the example policy has a size of 361 nodes. While it is useful to show the feasibility of transforming a policy in this way, the table itself is only 500 states large.

There are a few limitations to this method. One of the most prominent is that it requires human effort to determine the conditions necessary to use. Additionally, it does not learn the tree directly but rather it requires first obtaining a tabular policy,



```
+---------+    +---------+
|R: | : :G|    |R: | : :G|
| : : : : |    | : : : :█|
| : : : :█|    | : : : : |
| | : | : |    | | : | : |
|Y| : |B: |    |Y| : |B: |
+---------+    +---------+
```

Figure 6.4: Taxi environment states 283 and 183

which is not possible for all environments. The result is an approximation of a policy.

6.3 Learning a Tree with Pre-Specified Conditions

6.3.1 Method

This method learns a decision tree directly using reinforcement learning and human-specified conditions.

Find the algorithm for this method in Algorithm 3. Lines 1-4 describe the setup parameters. The condition functions C are the human-specified conditions. Lines 5-8 describe the creation of a mapping M of literal states $s_l \in S_L$ to abstract states $s_c \in S_C$. This mapping is determined by how each condition in C evaluates s_l (the “condition profile”). Lines 9-11 describe some subsequent setup before running the main algorithm in lines 12-16. Q-learning is run with a Q-table policy and a Tree policy, with state space S_C instead of S_L. Since the real observations will be in S_L, the mapping M is used to convert a literal state s_l to an abstract state s_c. The Q-table is updated after taking an action just as in standard Q-learning. The Q-table is never used directly to choose an action, however. Instead, a tree $\mathbb{T}$ is derived from the Q-table using ID3, and that decision tree is utilized during the exploit step to choose an action. At the end of training, $\mathbb{T}$ is the resulting decision tree policy.

This results in lower memory requirements, since only abstract states are tracked instead of tracking every state. When the algorithm learns information by taking an action from a particular literal state, it assumes the same result applies to other states in the abstract state. Although this method uses a table as part of its process, the actual policy is represented by a decision tree.

6.3.2 Results

We again use the taxi domain with the 19 human-specified conditions from Figure 6.3. 500 literal states were transformed into 370 abstract states.

We trained on 100,000 episodes with greedy-ϵ. We used an episode time-out of 100 timesteps (if the agent did not succeed in 100 timesteps, the episode would terminate). In the beginning of training, it would always timeout and penalties per episode ranged from 24 to 33.

At the end of the training, evaluating the resulting policy on 10 episodes showed an average timesteps per episode of 82.2 (it solved the task within 100 timesteps sometimes, and other times it timed out). The penalties incurred were reduced to 0. While this is far from optimal, it did learn. Extending training to one million

Algorithm 3: Procedure for Learning a Decision Tree with Human Specified Conditions in Realtime

```
1  Given an indexed, ordered set of condition functions $(c : s \rightarrow B) \in C$ such that
     $c(s)$ yields a boolean $\forall c, s$;
2  $s_l \in S_L \leftarrow$ set of literal states;
3  $a \in A \leftarrow$ set of actions;
4  $d \leftarrow$ maximum depth for the decision tree to be learned (can be unlimited, with
     finite conditions it will be a finite tree);
5  A vector of length $n_c$ (the number of conditions) can be produced for a state $s$
     by evaluating each condition $c$ at that state and setting the corresponding
     index in the vector to 0 for False or 1 for True. Call this vector the
     "**condition profile**" of a state.;
6  $s_l \in g \in G \leftarrow$ set of sets of states such that for a given grouping $g$, all $s_l \in g$
     have the same condition profile, no $s_l$ in different $g$ have the same condition
     profile, and each $s_l$ is distributed in exactly one set $g$.;
7  $s_c \in S_C \leftarrow$ set of abstract states formed by drawing one state from each $g \forall g$;
8  $M : (s_l \rightarrow s_c) \leftarrow$ a mapping that maps each $s_l$ to its corresponding $s_c$.;
9  $n_{sc} \leftarrow$ the size of $S_C$;
10 $F \leftarrow$ a $n_{sc}$ x $n_c$ features matrix where each row is the condition profile for each
     state $s_c$.;
11 $\mathbb{Q} \leftarrow$ a $n_{sc}$ x $n_a$ Q-table (where $n_a$ is the number of actions in the action space);
12 Initialize a decision tree $\mathbb{T}$ as a single leaf indicating a single action randomly
     chosen from $A$.;
13 Run a normal Q-learning algorithm on $(\mathbb{Q}, S_C, A)$ with the following
     adjustments:
14 • During exploitation, ignore $\mathbb{Q}$ and use $\mathbb{T}$ to choose the action to perform
     instead
15 • When updating Q, consider the actual state of the environment $s_l$, use $M$ to
     map it to $s_c$, and update the row of the Q-table Q corresponding to $s_c$
16 • When any row of Q is updated, IFF the column with maximum reward for
     that row changes, update $\mathbb{T}$ to be the decision tree that best fits the current $Q$,
     using $F$ as the features in the ID3 algorithm, with maximum depth $d$ if set.
17 At the conclusion of training, $\mathbb{T}$ represents the learned decision tree.
```

training episodes does not improve the average time to complete the task, suggesting that this is a limitation of the method, or at least of this method on this domain.

6.4 Summary

This chapter introduced a method that trained using a standard tabular method and transformed the policy after the fact. Then, it introduced a method that learns a decision tree without any intermediary representation.

There are scenarios where, at this point in time, a computer cannot determine what is or is not a relevant feature by which to make decisions, but a human can. In such a scenario, it may be the case that a human can posit possible features, but not determine an optimal policy in the manner that an RL algorithm can.

In these situations, techniques such as those introduced in this chapter become relevant. The techniques here presented involved decision tree policies. Humans declared possible conditions upon which the tree could split, and the methods used reinforcement learning to determine the optimal policy thereby.

Chapter 7

Explainable Reinforcement Learning via Decision Tree Policies: Conservative Q-Improvement

This chapter switches focus from constructing structured policies under human direction or learning with a combination of human input and autonomous learning to learning structured policies completely autonomously. It might be asked whether we should continue to care about structure if the policy is going to be learned completely autonomously. One of the benefits of a structured policy is how it allows a human to understand or interact with the policy.

Understanding a learned policy would be useful in several types of situations. First, it would be useful where safety is a concern, such as in vehicles or drones [55, 69]. It can also affect the degree of trust in a system. Doctors are less likely to trust decision-making aids they cannot explain, even if the model has actually discovered useful new information [13, 35]. Having the ability to perform a human check could increase trust in a model that was created in simulation but not yet exposed to the real world. There is the potential for harmful bias (i.e. sexism) in a learned model[14, 28], another area in which a human check could be useful. Allowing human review would also apply in situations where simulation is possible but testing the model in real life is extremely expensive (such as space exploration [9]).

A subsequent goal will be to have a robot autonomously learn a TAIG in order to solve a task. To move towards this end, this chapter keeps the focus on learning a policy in the form of a decision tree. A decision tree has conditional branches, which is a subset of TAIG functionality. We will improve upon an existing technique to learn a decision tree policy via reinforcement learning. Our method allows an agent to learn a decision tree using reinforcement learning where it makes decisions about on what features to split in the observation space.

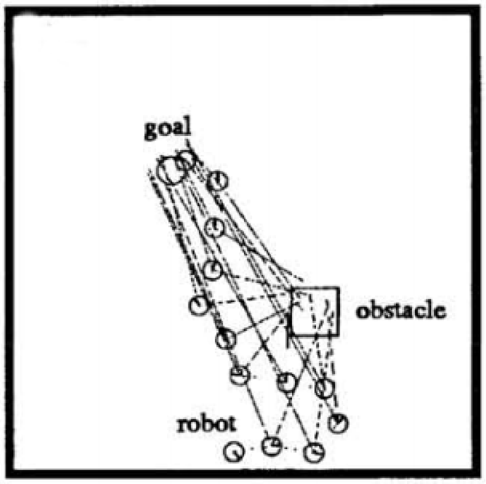


Figure 7.1: An image of Robot Navigation environment

Two environments are introduced in Sections 7.1 and 7.2, both of which will be used to test the methods. All environments used in this chapter are implemented in AI Gym from Open AI [19]. Section 7.3 provides background information on the previously mentioned "existing technique," the Partial Conservative Q-Improvement (pCQI) method. Section 7.4.1 introduces Conservative Q-Improvement (CQI), which improves upon pCQI by adding fine-splitting capabilities. Section 7.4.2 introduces Conservative Q-Improvement 2, which improves upon CQI by adding Feature Combinations. We test these methods on the environments discussed in this chapter, comparing them against each other as well as against a baseline reference method. The basic results are described in Section 7.5.1. Since one of the attributes of the CQI/CQI2 methods is the ability to choose for optimizing for different goals (i.e. overall reward, size), Section 7.5.2 delves into how to go about tuning hyperparameters to achieve these tradeoffs.

7.1 Modified RobotNav Environment

We use the modified RobotNav environment described in [80] to test the methods. In this 2D environment, there is a robot, a goal location, and one or more obstacles/holes. The robot must navigate to the goal while avoiding the obstacles/holes, as shown in Figure 7.1[1]. There are three actions in the action space: i) move directly towards goal, ii) move perpendicular to direct-line-to-goal (right), iii) move perpendicular to direct-line-to-goal (left).

[1]Image taken from [62, 80]

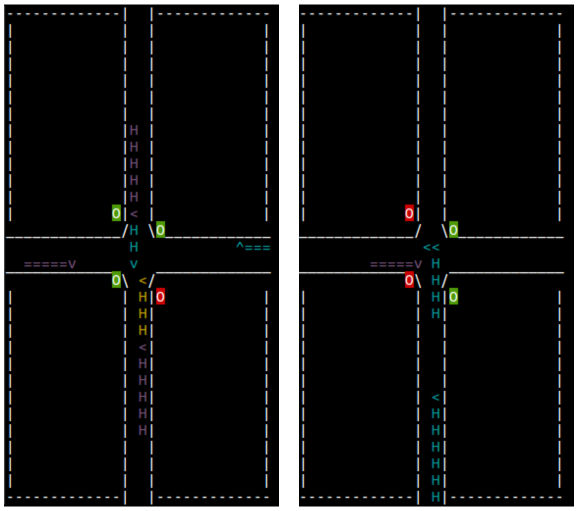


Figure 7.2: Two examples of rendering of the vehicle intersection environment

Additional modifications can be made to the environment to yield, effectively, multiple similar environments. Using the 4-length state space as in [80] of [i) angle between agent and goal, ii) distance from agent to goal, iii) angle between agent and hole/obstacle, iv) distance between agent and hole/obstacle], is henceforth called "Robot Navigation A". Using the alternate 5-length state space [i) agent position x, ii) agent position y, iii) hole/obstacle position x, iv) hole/obstacle position y, v) boolean (0 or 1) indicating whether the agent was closer to the goal than the hole/obstacle was], will henceforth instead be termed "Robot Navigation B."

The code for the Robot Navigation environment can be found at `https://github.com/AMR-/CMU-Structured-Representations-For-Robots/blob/master/RobotNavigation/RobotNavigation.py`.

7.2 Vehicle Intersection Environment

Additionally, we present a new OpenAI Gym "Toy Text"-compatible environment: "Vehicle Intersection". We also provide the code for this configurable and difficult environment.[2] It is described in some detail here, with additional documentation in Appendix C. This is an additional environment that we use to test the methods.

In Vehicle Intersection, vehicles approach a four-way intersection from the four two-lane roads. Each approach to the intersection has a stoplight that can be red or green. The agent controls these four traffic lights, and actions involve turning the lights red or green.

[2]Find it on GitHub at `https://github.com/AMR-/gym_vehicle_intersection`

The vehicles move at a fixed speed when unimpeded. If they are blocked by a vehicle in front of them going in the same direction, they stop. If they are blocked by a vehicle in front of them going in a different direction, there is a collision. If a vehicle approaches the edge of the intersection and encounters a green light, it continues moving. If it approaches the edge of an intersection and encounters a red light, it stops and waits until the light is green, at which point it moves forward.

Additionally, each car has an attribute called “turn signal” which can be one of three values: “none,” “left,” or “right.” This indicates what direction it will turn at the intersection. Vehicles come in different lengths. Movement and time are discrete.

Each episode, there are a set number of vehicles that will spawn on the map over the course of the episode. The episode completes when either this number of vehicles has safely exited the intersection or when there is a collision between any two vehicles.

The agent receives a penalty of -1 when a vehicle is forced to wait at a red light (per vehicle, per timestep). A vehicle stuck behind another vehicle waiting at a red light also incurs this penalty. The agent receives a penalty of -1000 when a collision occurs. An optimal episode has, at most, a reward of 0 (which may not always be possible depending how vehicles spawn). The goal becomes that of reducing waiting time for vehicles while also avoiding collisions.

The environment has a `render()` method which shows a visualization in colored ASCII characters. Find an example in Figure 7.2. The shown rendering includes the default road length of 14. The lights are situated near the intersection, to the right of each road, and display red or green. In both examples, at least one vehicle is waiting at a red light while another passes through the intersection. At right is a vehicle in the midst of turning left through the intersection, as a vehicle approaching from the left-road waits at a red light (when it turns green, it will turn right). At the front of each vehicle an arrow indicates the turn signal.

There is no acceleration in this environment. Vehicles move or they don’t. There is a delay in vehicle movement of one timestep when a red light turns green. (When an action is taken, vehicles move first, and then the lights change.)

Action spaces are discrete options. In this work the **SetByLight** action set is used (for a discrete action space of size 9) instead of the default.

The following are some of the customizable options for the environment and their defaults (for a full list, see the documentation in Appendix C):

- action_set - which of the action sets to use (ToggleRoad(), ToggleLight(), SetByRoad(), SetByLight(), SetExplicitly()) (see full documentation in Appendix C or online for details). default: ToggleRoad()
- road_length - length of the road before the intersection. (intersection length is 2.) default: 14
- use_turn_signals - boolean. if False, cars only go straight. default: True
- max_vehicles_on_map - when this number of vehicles are on the map, do not

spawn additional vehicles until at least one exits the intersection. (Note: This parameter affects the size of the observation space.) default: 8

- vehicles_to_spawn_per_episode - the number of vehicles to spawn per episode. affects episode length. default: 10
- waiting_penalty. default: -1
- collision_penalty. default: -1000

The following equation gives the upper limit of the size of the observation space (some states are unreachable):

$$16 \cdot V \cdot (4 \cdot (L + 2) \cdot R) \tag{7.1}$$

where V is the maximum number of vehicles allowed on the map, L is the road length, and $R = 3$ if turn signals are used, $R = 1$ otherwise.

In this thesis, Vehicle Intersection has **max_vehicles_on_map** set to 2.

7.3 Partial-CQI

This thesis builds upon what is termed the Partial Conservative Q-Improvement (pCQI) method, a joint work with Topin, Jamshidi, and Veloso which is a part of the Conservative Q-Improvement method (CQI) [80]. We discuss pCQI here in some detail.

The pCQI method is a technique for learning a decision-tree-style policy using RL. The tree is initialized with a single leaf node, representing a single abstract state. Over time, it creates branches and nodes by replacing existing leaf nodes with a branch node and two child leaf nodes.

The tree is split only when the expected discounted future reward of the new policy would increase above a dynamic threshold. This minimum threshold decreases slowly over the course of training and resets to its initial value after every split.

The high-level algorithm is shown in Algorithm 4. At a given timestep, the environment will be in state s_t. This will correspond to some leaf node L. When there is a single leaf node, states correspond to this node. When there are multiple leaf nodes, the tree must be traversed to determine the corresponding L (each branch node having a boolean condition that operates on s_t, indicating which of two children to traverse).

Once L is identified, an action a_t is chosen. If exploring, a random action is taken. If exploiting, an action is chosen based on the highest Q-values on that leaf. Executing the action yields a reward r_t and next-state s_{t+1}. A series of updates are then performed.

Each node tracks information on Q-values for each action and visit frequency for the node. The Q-values on leaf L are updated using the standard Bellman equation update. Each node keeps track of how often it has been visited. Each leaf node maintains a history of possible splits (one per dimension, splitting on the halfway point of each dimension). These are potential means of converting the leaf node into a branch node with two child nodes. These estimated child nodes have hypothetical visit frequencies and Q-values-per-each-action, which are updated at each update step where the agent is in a state corresponding to a given node.

Algorithm 4: Partial Conservative Q-Improvement

```
1  $\mathbb{T} \leftarrow$ initial tree is a single leaf node;
2  $H_S \leftarrow$ the starting threshold for what potential $\Delta Q$ is required to trigger a split;
3  $h_S \leftarrow H_S$;
4  $D \leftarrow$ the decay for $H_S$ and $h_S$ values;
5  for number of episodes do
6      while episode not done do
7          $s_t \leftarrow$ current state at timestep $t$;
8          $L \leftarrow$ leaf of $\mathbb{T}$ corresponding to $s_t$;
9          $a_t, r_t, s_{t+1} \leftarrow$ TakeAction($L$);
10         UpdateLeafQValues($L, a_t, r_t, s_{t+1}$);
11         UpdateVisitFrequency($Tree, L$);
12         UpdatePossibleSplits($L, s_t, a_t, s_{t+1}$);
13         bestSplit, bestValue $\leftarrow$ BestSplit($\mathbb{T}, L, a_t$);
14         if $bestValue > h_S$ then
15             SplitNode($L$, bestSplit);
16         else
17             $h_S \leftarrow h_S \cdot D$
18         end
19     end
20 end
```

Then, the possible splits are checked to determine which split would yield the most *split value*, where split value is increase-in-expected-reward moderated by visit frequency. If this value is beyond a threshold h_S, the split occurs. Otherwise, this threshold is decreased slightly, by a decay value D. Splits can occur at any timestep. h_S loosely affects the rate at which splits will occur.

In contrast to the methods described in Chapter 6, pCQI-style methods do not require a human to determine ahead of time which conditions should be considered for a split. The method makes that determination itself. What it does require is that the environment state space be represented as a multidimensional vector, with elements having some semantic meaning.

Please find a diagram of how pCQI (and derivative methods) work in Figure 7.3.

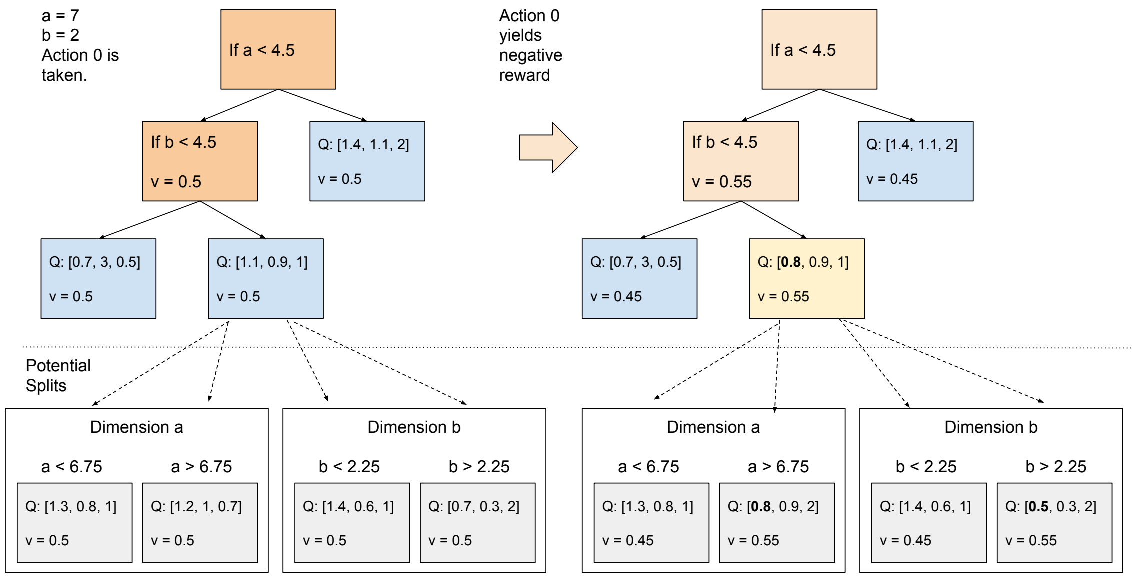


Figure 7.3: Example of an update to the policy tree in pCQI

This shows a tree in the process of being constructed. In this example, the state space has two features, feature a and feature b, each of which can take values from 0 through 9. There are three possible actions. At left the tree is shown at a certain timestep, and at right after the action for that timestep has been taken. At left, see the two branch nodes in orange and three leaf nodes in blue. Each branch node has a condition, and every node aside from the root node has a visit frequency count. Truthfully satisfying the condition indicates the rightward branch. Let it be assumed that the current state is ($a = 7$, $b = 2$). Since $a \not> 4.5$, the left branch is traversed, and then since $b < 4.5$, the right branch is traversed. The maximum Q-value at this leaf is at action 0, and so action 0 is taken. In this example, a negative reward results. It can be seen on the tree on the right how the visit frequency metrics are updated. The path from the root node to the leaf node (highlighted) has visit frequency increased and sibling nodes to affected nodes have their visit frequency decreased. The Q-values on the leaf node are updated.

Each leaf node also has a number of "potential splits." Displayed in the diagram are the two possible nodes that could be created per dimension. These potential nodes have their own visit frequencies and Q-values which are updated *as if* they existed on the main tree. When one of these splits seems likely to yield a better policy than the current one (according to a formula specified in the source paper), a split is executed, and potential nodes become real nodes.

7.4 CQI and CQI2: Method

There are two main innovations to pCQI discussed in this thesis. The first (multiple splits per dimension), changes partial CQI (pCQI) to full CQI (CQI). The second (feature combination functions) yields CQI version 2 (CQI2).

7.4.1 CQI: Multiple Possible Split Locations Per Dimensions

The innovation in Conservative Q-Improvement (CQI) is to allow multiple possible split locations per dimension. The pCQI algorithm stores possible splits, where each possible split was a single division of an existing dimension precisely in half (according to the stated bounds of the dimension's values). In CQI, the same dimension of the space can be split multiple times in multiple locations, which can enable reaching the optimal policy with less nodes. A configurable hyperparameter n_S is added, indicating the "number of splits." The n_S value controls how many split-points are considered as possibilities in each dimension. (pCQI is equivalent to CQI with $n_S = 2$, the minimum valid value for this hyperparameter.)

In pCQI, for situations where the optimal split is not the halfway point between

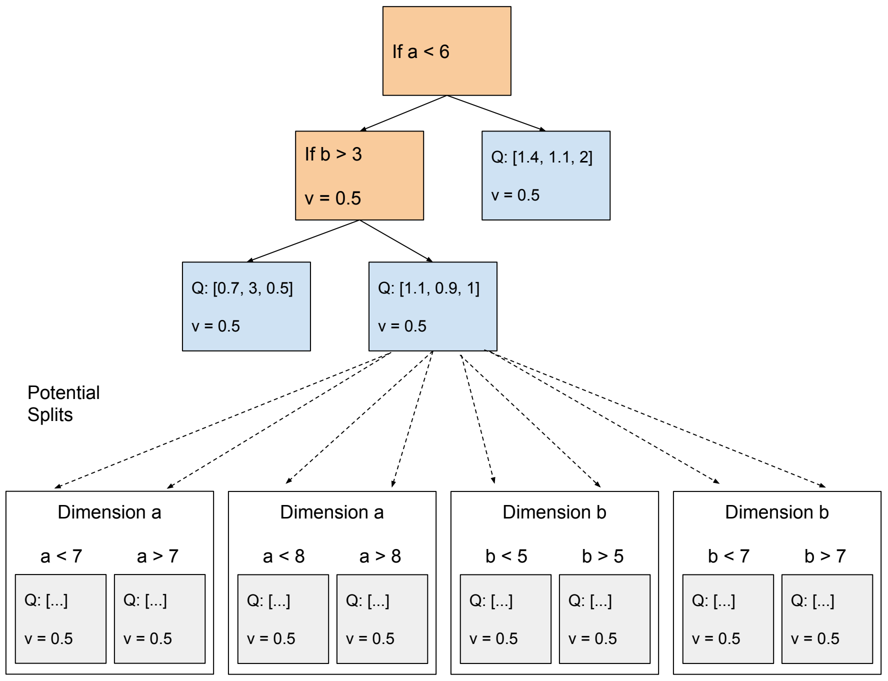


Figure 7.4: Example of the multiple Split objects per dimension in CQI

the lower and upper bounds of a feature value, multiple nodes would have to be used to describe this situation. CQI and CQI2 can handle such situations in a single node. For example, with CQI2 and $n_S = 8$ or a multiple of 8, a dimension with optimal first-split location of one-eighth of the distance from lower bound to upper bound could be split on that optimal location. In contrast pCQI would have to use three nodes in succession, first splitting on the halfway point, then on the quarter-way point, and only then on the eighth-way point.

The algorithm for CQI is similar to pCQI. The main difference is an initialization and in the SplitNode function, shown in Algorithm 5. SplitNode takes a leaf node and turn it into a branch node with two children leaf nodes.

Algorithm 5: SplitNode
Splits leaf node N into nodes B, L, R

```
SplitNode($N$, bestSplit):
Let $B$ be an internal branching node.;
Let $L, R$ be left and right children of $B$, respectively.;
$B[v] \leftarrow N[v]$;
$B[m] \leftarrow$ bestSplit$[m]$;
$B[u] \leftarrow$ bestSplit$[u]$;
for $N \in \{L, R\}$ do
    $N$[Splits] $\leftarrow$ { Add a set of left-right split pairs, one for each dimension and split.}
end
$L[v] \leftarrow$ bestSplit[left]$[v]$;
$L[Q] \leftarrow$ bestSplit[left]$[Q]$;
$R[v] \leftarrow$ bestSplit[right]$[v]$;
$R[Q] \leftarrow$ bestSplit[right]$[Q]$;
Return $B$, $L$, $R$
————————————
Hyperparameters:
$Q_{init} \leftarrow$ default Q-value;
$n_S \leftarrow$ the number of splits to check for in each dimension;
```

In both cases, a leaf node has the following attributes: visit frequency v, a mapping of actions to Q-values ($Q : (a \rightarrow q)$), and a list of `Split` objects `Splits`. Each `Split` object contains information about potential splits (one each per potential split), and has dimension number m, value to split on u_p, and Q-value and visit frequency for both of the potential children (`left`.q, `left`.v, `right`.q, and `right`.v). In pCQI each `Splits` list will contain one `Split` per dimension, and u_p will be the halfway point between the lower and upper bounds of `Split`$[m]$. In CQI, there will be $(n_S - 1)$

number of `Split` objects per-dimension per `Splits` list. The n^{th} `Split` for dimension m will contain a u_p with value $u_p = \frac{n}{n_S} \cdot (m_U - m_L) + m_L$ for $n = 1$ to $(n_S - 1)$, where m_U and m_L are the upper and lower bounds of dimension m, respectively. In pCQI, $\texttt{length}(\texttt{Splits}) = 2 \cdot M$, and in CQI, $\texttt{length}(\texttt{Splits}) = n_S \cdot M$, where M is the total number of dimensions in the state space.

The `Splits` list is initialized in the root node at the beginning of training, and is initialized in new nodes created by the SplitNode function.

Find an example diagram of this situation in Figure 7.4. In this example, the amount of splits has been increased to 3, such that there are two split-points per dimension instead of one. Node traversal, splitting, and metric updates all occur in the same manner as in pCQI.

7.4.2 CQI2: Feature Combinations

In some cases it is useful to be able to choose an action based on multiple features, or how one feature relates to another. To this end, Conservative Q-Improvement 2 (CQI2) enables the ability to use Feature Combination Functions (FCFs) to combine features in the state space. In our technique, FCFs are used to expand the real state space into a larger conceptual state space with additional dimensions that are functions of real features. For example, a new dimension m_3 might be created that contains the sum of the values in dimensions m_1 and m_2.

At runtime, CQI2 takes in a list of FCF's to use. An FCF is a tuple as shown in Equation 7.2.

$$\begin{aligned}\text{FCF} = &< F_F, F_B, \text{commutative} > \\ &F_F : (x_{\text{val}}, y_{\text{val}}) \rightarrow z_{\text{val}} \\ &F_B : (x_{\text{low}}, x_{\text{high}}, y_{\text{low}}, y_{\text{high}}) \rightarrow (z_{\text{low}}, z_{\text{high}}) \\ &\text{commutative} \in \{\text{True}, \text{False}\}\end{aligned} \tag{7.2}$$

where all x_i, y_i, z_i values are scalar values of features (or lower/upper bounds), F_F is the combination function (such as summation), F_B is a function used to calculate the bounds of new dimension z based on the bounds of x and y, and `commutative` is a boolean indicating whether F_F is commutative.

The FCFs are used to expand the state space. The state expansion process is described in Algorithm 6.

A similar process is used to find the bounds of the expanded state space from the real state space bounds (but using the F_B instead of F_F).

The number of states in the expanded state space E is given by $E = \left((2 \cdot N_{\text{ncomm}} + N_{\text{comm}}) \cdot \binom{N_R}{2} + N_R\right)$, where N_{ncomm} and N_{comm} are the number

Algorithm 6: AugmentState augments multidimensional state $s \in \mathcal{R}^M$ into expanded multidimensional state $s' \in \mathcal{R}^{M'}$ using a list of FCF's A

```
AugmentState($s$, $\mathcal{A}$):
Let $\mathcal{P}$ be the ordered set of all combinations of pairs of dimensions $m_i$ in $s$
  (ordered by a consistent process);
$s' \leftarrow s$;
for $(m_i, m_j) \in \mathcal{P}$ do
    for $c \in \mathcal{A}$ do
        $F_F \leftarrow c[F_F]$;
        $m_{\text{new}} \leftarrow F_F(m_i, m_j)$;
        Append value of $m_{\text{new}}$ to $s'$ as a new dimension;
        if $c[commutative]$ then
            $m_{\text{new}} \leftarrow F_F(m_j, m_i)$;
            Append value of $m_{\text{new}}$ to $s'$ as a new dimension;
        end
    end
end
Return $s'$ as the expanded state;
```

of non-commutative and commutative FCFs, respectively, and N_R is the number of real states.

The environment simulator does not need to be changed, new states are expanded from real states at every step (although a cache is used to speed up processing of repeat states, if any). Nor does the CQI2 algorithm need to be changed in any way. From the perspective of the underlying CQI2 algorithm, the combination states are just like any other state, and they can be split on in the same manner.

Indeed, one of the benefits of this concept is that in theory it could be applied to expand the state space of any multidimensional environment in front of any learning algorithm. It is, however, of particular benefit to CQI-style learners to have additional useful bounded scalar dimensions from which to choose.

7.5 CQI and CQI2: Results

This section presents a comparison of the Pyeatt Method (PM)[3], pCQI, CQI, and CQI2. We evaluate these methods on three environments: Robot Navigation A, Robot Navigation B, and Vehicle Intersection ($V = 2$).

[3]The Pyeatt Method was discussed in Section 2.3.

In Robot Navigation A and B, partial Conservative Q-Improvement and derivatives create trees with greater average reward per episode than those created with the Pyeatt Method. Also in these two environments, trees created via *CQI methods have an order of magnitude fewer nodes than those created via PM. In Robot Navigation A and B, CQI improves upon pCQI in the metrics of both policy size and reward. In VehicleIntersection, CQI2 demonstrates an improvement over pCQI and CQI in both policy size and reward.

7.5.1 Direct Comparison

To fairly compare methods, a grid search was performed for each method to determine the best hyperparameters. The bounds of the tested values for each hyperparameter for each environment are shown in Tables D.1 and D.2 in Appendix D. The visit decay and split threshold decay parameters for pCQI, CQI, and CQI2 are held constant at 0.999 and 0.9999, respectively. All of the best hyperparameter configurations fall within the ranges searched.

In our runs of CQI2, the following functions are used as FCFs: i) summation $(x + y)$, ii) equality boolean $(x == y)$, iii) subtraction $(x - y)$, and iv) greater than boolean $(x > y)$. The first two are commutative and the last two are not. (So in Algorithm 6, A is of size 6. For Robot Navigation A, the real state space is of size 4 and the expanded state space is of size $E = 40$. For Robot Navigation B, the real state space is of size 5 and the expanded state space is of size $E = 60$. For Vehicle Intersection, the real state space (when setting the number of vehicles on the map to 2) is 12, and the expanded state space is of size $E = 408$.

In all cases a greedy-ϵ of $\epsilon = \max(0.05, 1 - \text{step}/d)$ was used.

For Robot Navigation A, three million $(3 \cdot 10^6)$ training steps were used, with $\gamma = 0.8, d = 400,000$. For Robot Navigation B, two million $(2 \cdot 10^6)$ training steps were used, with $\gamma = 0.8, d = 300,000$. For Vehicle Intersection, one million $(1 \cdot 10^6)$ training steps were used, with $\gamma = 0.9, d = 200,000$.

In all cases, 10 trials were performed of the above process at each configuration and the results were averaged. During each trial, the final policy was evaluated by averaging the results of the episodes over the course of 50,000 steps.

Find a table of results of methods in environments at optimal parameters in Table 7.1. In the "Method" column, '(R)' indicates hyperparameters optimized for reward and '(S)' indicates hyperparameters optimized for size (the smallest tree that has a higher reward than a previous method, unless otherwise noted). The specific hyperparameters associated with each optimal point are noted in Table D.3

Env:	**Robot Navigation A**				**Robot Navigation B**			
	Tree Size		**Avg. Reward/Ep.**		**Tree Size**		**Avg. Reward/Ep.**	
Method	Avg.	Std. Dev.	Avg.	Std. Dev	Avg.	Std. Dev.	Avg.	Std. Dev
PM	1917.8	9.10	-48.714	51.58	493.8	3.79	-61.29	45.97
pCQI (R)	7.8	1.40	-18.800	0.25	20.2	3.79	-20.10	0.31
pCQI (S)	7.0	0.00	-18.906	0.18	20.2	3.79	-20.10	0.31
CQI (R)	43.2	5.03	**-18.753**	0.62	22.2	4.34	**-19.99**	0.31
CQI (S)	**7.0**	0.00	-18.766	0.30	**18.6**	3.98	-20.10	0.52
CQI2 (R)	47.8	2.86	-21.077	2.98	181.8	46.27	-22.82	4.99
CQI2 (S)	31.6	0.97	-37.681	45.87	148.8	35.60	-23.85	5.27

Env:	**Vehicle Intersection**			
	Tree Size		**Avg. Reward/Ep.**	
Method	Avg.	Std. Dev.	Avg.	Std. Dev
PM	**201.0**	0.00	-747.551	32.54
pCQI (R)	12403.4	629.55	-717.211	117.94
pCQI (S)	9044.4	385.26	-733.602	117.10
CQI (R)	21941.6	587.88	-693.113	135.30
CQI (S)[4]	5179.8	459.83	-738.748	59.47
CQI (S)[5]	11897.2	519.44	-716.942	86.56
CQI2 (R)	19345.6	1083.62	**-468.268**	46.39
CQI2 (S)[6]	**1708.8**	164.68	-710.316	117.69
CQI2 (S)[7]	**3491.4**	248.56	-641.918	117.90

Table 7.1: Comparison of RL decision tree policy methods across environments

in Appendix D.

In Figures 7.5 and 7.6, find, for selected methods, a plot of the size of the tree policy *during* training vs. the reward that it has achieved at that size. (The number of training steps is constant at 10^6, and the lines end at different sizes since different policies yield different final tree sizes.)

Across all three environments, the CQI method is shown to be capable of achieving a higher reward than either the Pyeatt Method or pCQI. Additionally, in all environments, the CQI method is capable of producing a smaller tree for a given minimum reward threshold than is the pCQI environment (or in the case of Robot Navigation A, achieving a better reward for the same minimum tree size). In Robot Navigation A and B, even the larger tree associated with the highest reward value is smaller than the tree produced by the Pyeatt Method. In the Vehicle Intersection

[4]Optimized for size with respect to Pyeatt. This is the smallest size tree for CQI whose reward surpasses the highest reward the Pyeatt Method can achieve at any size.

[5]Optimized for size with respect to pCQI. This is the smallest sized tree for CQI whose reward surpasses the highest reward the pCQI method can achieve at any size.

[6]Optimized for size with respect to pCQI. This is the smallest sized tree for CQI2 whose reward surpasses the highest reward the pCQI method can achieve at any size.

[7]Optimized for size with respect to CQI. This is the smallest sized tree for CQI2 whose reward surpasses the highest reward the CQI method can achieve at any size.

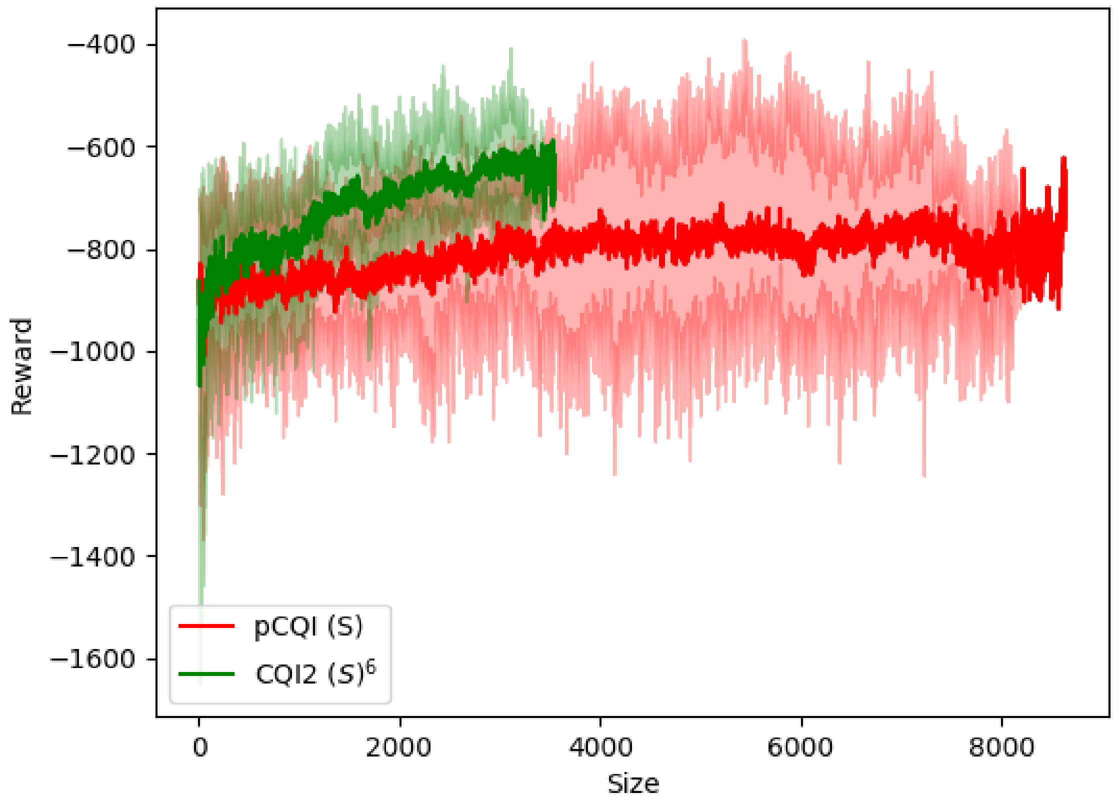


Figure 7.5: Comparison of reward vs size during training for pCQI vs CQI2 on Vehicle Intersection

Environment, even though the Pyeatt Method has a smaller-size/smaller-reward tree compared to the *CQI family, CQI is able to surpass Pyeatt (achieve a higher reward than that of which Pyeatt is capable) with a smaller tree than the tree with which pCQI is able to surpass Pyeatt. Here CQI is also able to beat pCQI's best score while maintaining a smaller size than the size of the tree for pCQI's best-scoring tree. All of this is enabled by the addition of more than two split possibilities per node.

CQI2 is capable of achieving a higher reward than the Pyeatt Method in all environments. In Robot Navigation A and B, CQI2 does so while achieving a smaller size tree overall, even at reward-optimized parameters. CQI2 achieves a significantly higher reward than all other methods in the Vehicle Intersection environment. Also in this environment, CQI2 is capable of producing a smaller final tree for a given minimum reward threshold when compared separately to each of the pCQI and CQI methods. This demonstrates what can be achieved with FCFs.

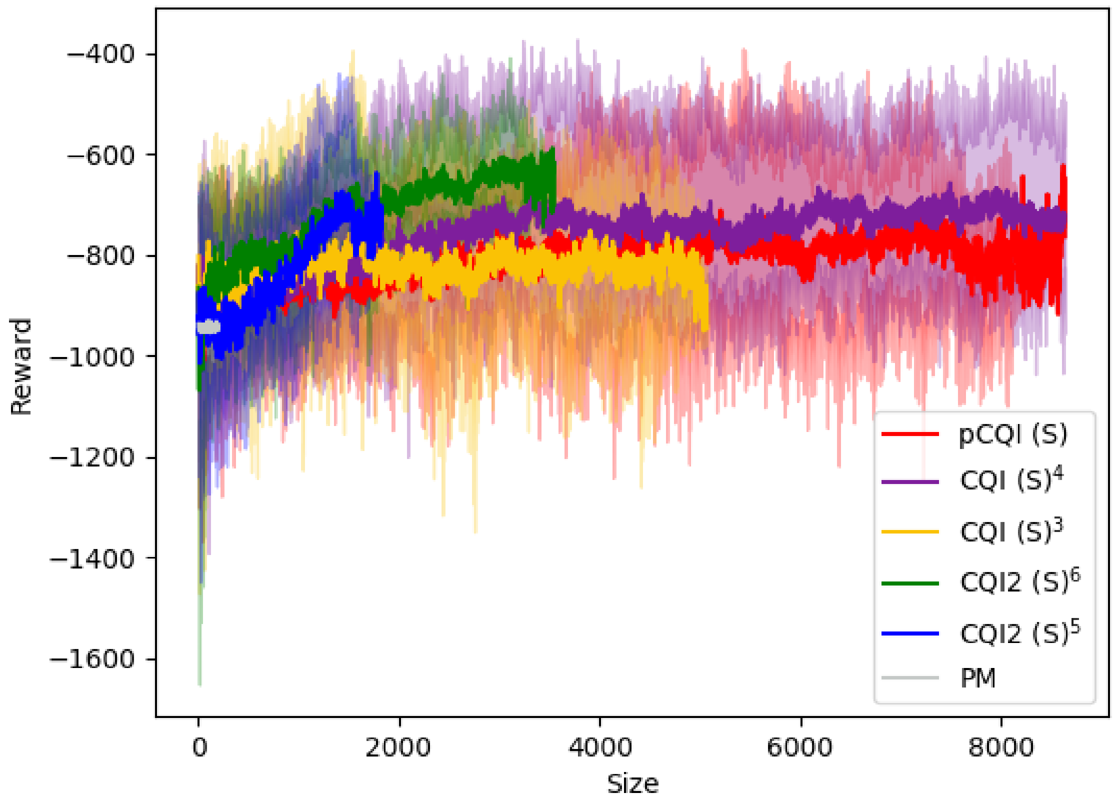


Figure 7.6: Comparison of reward vs size during training for various methods on Vehicle Intersection

We also speculate that longer training times would enable reaching higher rewards and smaller trees for the CQI2 method across all environments. Every time that the number of training steps was increased, both reward and size metrics improved for CQI2. (The other methods had small improvement, if any, beyond the 500k training steps point.) The idea that more training would be specifically helpful to CQI2 is bolstered by Figures 7.5 and 7.6, where CQI2 lines seem to be on an upward slope while others have plateaued. Testing beyond the training steps noted here was not possible due to real-world resource and time constraints. One of the reasons that CQI2 takes so much longer than other methods to achieve the optimal policy is that the state space is exponentially larger, and so can take a correspondingly larger amount of time to explore, even with the savings of inferring information about abstract states.

Another benefit of the CQI2 and CQI methods are their ability to make a trade-off

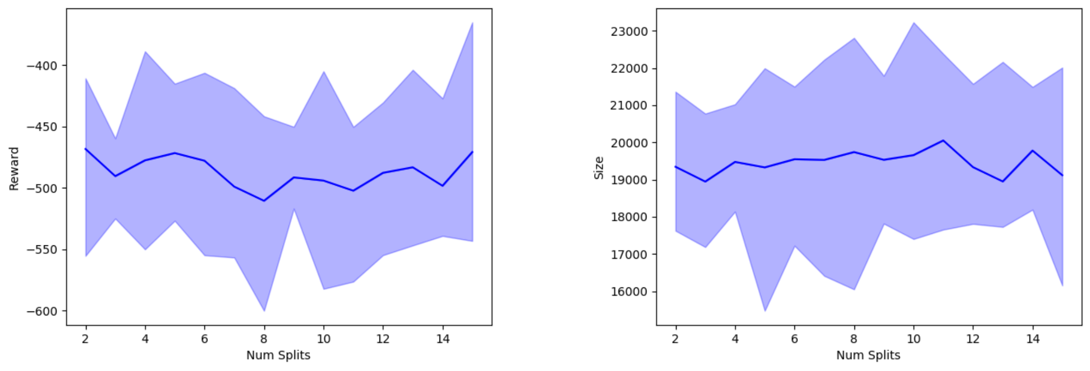


Figure 7.7: Analysis of the effect of the n_S CQI/CQI2 hyperparameter

between optimizing for size or optimizing for reward. This is suggested by Table 7.1 and discussed further in the next section.

7.5.2 Parameter Sensitivity

The previous section referred to the ability to optimize either for the size of the policy or for the highest reward. This section discusses the ability of CQI and CQI2 to tradeoff between these two optimalities, and how tuning the hyperparameters specific to this method affects the outcome. This will be useful to practitioners seeking to use this method.

In Figure 7.7, find graphs showing the effect of tuning the n_S ("number of splits") parameter while other hyperparameters are kept at optimal values for runs of Vehicle Intersection with CQI2. The line is average values, and the shaded areas show minimum and maximum values. It can be seen that in general, there is no direct correlation between this parameter and better policies, whether in the case of change in policy size or reward value.

A better or worse n_S value is likely policy specific. Sometimes the ability to split at certain locations is better than others.

However, having this as a tunable parameter itself clearly improves the ability of the overall search to yield a superior policy on either desired metric. This is shown in the previous section, where across all environments, CQI had a higher maximum reward than pCQI, and lower minimum achievable size for a given level of reward achieved.

There is no guarantee that moving the value of n_S in one direction will have a particular effect. Thus, in practice, searching widely is useful to determine which value is best, as it can result in a superior policy.

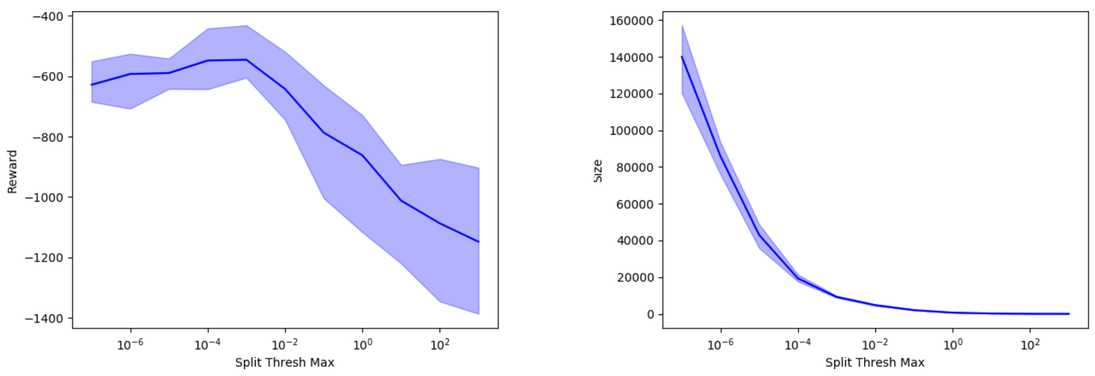


Figure 7.8: Analysis of the effect of the H_S CQI/CQI2 hyperparameter

In Figure 7.8, find graphs showing the effect of tuning the H_S parameter while other hyperparameters are kept at optimal values for runs of Vehicle Intersection with CQI2. The line is average values, and the shaded areas show minimum and maximum values.

It is very clear that increasing H_S results in a smaller size policy, and decreasing results in a larger size policy (when number of training steps are fixed). This is intuitive, since a higher H_S will cause the algorithm to wait longer before making a split. Waiting longer to perform a split means that the earlier splits are better informed and therefore higher quality. When this happens, it is more difficult to improve the policy further, so it takes more steps for potential splits to reach the split threshold. Larger values can yield smaller trees.

The graph of behavior of reward is also very interesting, as two trends are seen on either side of the clearly optimal reward. First, as H_S is increased, reward increases[8]. This makes sense because when the algorithm waits longer before making a split, the split will be made at a potentially more informed location, leading to a higher-scoring overall policy.

Second, past a certain point, reward decreases. This is because if the algorithm waits too long to make a split, then in a finite series of training steps there may not be time to fully develop a successful policy (too long an interval between splits means that it is impossible to split enough times to create a successful policy).

Potentially, training time could allow for smaller trees without sacrificing final reward—as long as the training time is sufficient for the chosen H_S, the final policy will have sufficient time to train and avoid the "cliff" that results in poor policies.

Thus, during a tuning procedure, a user of this method could choose to increase

[8]Although it may not be as stark visually as the subsequent decrease, from $H_S = 10^{-6}$ to 10^{-4} there is a marked increase in average reward.

H_S to search for smaller policies, or decrease H_S if they wonder if a larger policy might better capture a higher reward policy (if they thought current values to be on the right side of the reward-optimal curve).

7.6 Summary

This chapter discusses various related methods for conducting reinforcement learning to solve tasks where the policy is in a decision tree format. It describes the Robot Navigation and Vehicle Intersection environments, the latter of which is itself a contribution of this thesis. It introduced the CQI and CQI2 methods, which extend pCQI.

This chapter shows how CQI and CQI2 outperform pCQI and PM. In all cases tested, the *CQI family yields greater reward than PM. CQI can reach smaller trees than pCQI for the same or greater reward in all environments tested, and in the environment with the largest state space tested, CQI2 reaches smaller trees with greater reward compared to both pCQI and CQI. Additionally, one of the relevant aspects of the CQI/CQI2 methods is the ability to target a high reward policy or a small size policy above a certain reward threshold. Achieving this requires some understanding of how the hyperparameters affect the final policy, and this detailed understanding is presented and demonstrated here as well.

The CQI and CQI2 methods contribute to the literature on explainable RL. Due to their decision tree structure. they allow a robot to autonomously learn policies that are interpretable.

Chapter 8

Learning a TAIG-style Policy with Reinforcement Learning

This chapter builds on the decision-tree-policies-via-RL methodology from Chapter 7 to learn a TAIG-style policy with WHILE loops.

Earlier thesis chapters demonstrated TAIG's power as a policy structure. Chapters 3, 4, and 5 discuss humans creating a TAIG through programming or interactive instruction.

If a robot were able to learn such a structure autonomously, it would be of great benefit. It would truly allow for human-robot collaboration. Then, tasks could be jointly learned and instructed. A learned task could be modified by a human, and an instructed task could be improved upon by an AI agent. It would represent a symbiosis of machine and human intelligence.

It would be even more of a challenge to build not just a TAIG, but one with WHILE-loops, a typically tricky control structure to learn autonomously.

This chapter achieves these goals, bringing this vision of future that much closer. Section 8.1 provides a high-level overview of the strategy for achieving this and the rationale behind it. Section 8.2 delves into the details of the method for building a TAIG via RL. Section 8.3 notes a couple of limitations of the method. Section 8.4 proves the feasibility of the method, showcasing TAIGs built via this RL method for the Robot Navigation B environment.

8.1 Overview of Approach

Chapters 6 and 7 introduce various methods to learn a decision-tree style policy through reinforcement learning. The branches of a decision tree are like IF-ELSE clauses in a TAIG, with one crucial difference. The decision tree policies of Chapter

7 indicate a decision-making process to be performed at every timestep, whereas a TAIG is a sequence of actions representing the entire task.

One can trivially transform a decision tree policy into a TAIG by surrounding the decision tree with a WHILE loop with the condition "while task is not complete". Nothing would be gained from doing this in terms of the actual performance of the task itself, but it allows us to think of the policy as a TAIG. The question can then be asked, "Can this policy be transformed into a more informative TAIG?"

One means of doing this would be to consider a task as a series of subtasks. This series of subtasks can each be represented by WHILE loops connected in series, where the policy in each subtask can be a decision tree. (Naturally, this formulation is only applicable to some types of tasks.)

The question then becomes, "Can this single tree be split into multiple policies across multiple WHILE loops?"

The following section addresses this question as it discusses a method that can be applied to a particular kind of two-stage task. It will learn a decision tree style policy through reinforcement learning, and then derive two trees from that single tree, placing each tree inside of a WHILE loop, thus autonomously learning a TAIG.

8.2 Methodology

There are a few prerequisites for this method to be applicable:

- The task must have two stages
- These two stages must correspond to a feature that exhibits the following behavior: constant at some value in the first stage, and constant at some different value in the second stage
- The above feature must be the split-upon feature in at least one branching node of a single-decision-tree-style successful policy

If the prerequisites are not satisfied, the method will not fail, but it will either not produce a policy with sequential While-loops (leaving the policy in single-decision-tree form) or it will produce two while-loops with the same decision tree in each (a meaningless separation).

The method has three parts:

1. Build a single-decision-tree style policy
2. Determine if there is a feature that separates the task into two stages, and if so, what are the properties of the feature's change between stages
3. Separate the single decision tree into two decision trees, place each decision tree into an appropriately configured WHILE loop, and place the WHILE loops in series with each other

This sequence is described in a more formulaic manner in Algorithm 7. Line 1 corresponds to step 1, line 2 corresponds to step 2, and lines 3 through 13 correspond to step 3.

Algorithm 7: The three steps of the procedure to build a TAIG-style policy with two while loops in sequence

```
1  $\pi_0 : (s \rightarrow a) \leftarrow$ single-decision-tree policy that maps state $s$ to action $a$;
2  $D$, comp, $u_D \leftarrow$ **FindSubtaskFeature**($\pi_0$);
3  **if** *some value returned for* $D$ **then**
4      **if** *comp* **then**
5          $W1 \leftarrow$ a condition primitive with condition $s[D] < u_D$;
6      **else**
7          $W1 \leftarrow$ a condition primitive with condition $s[D] >= u_D$;
8      **end**
9      $W2 \leftarrow$ a condition primitive with condition “task is done”;
10     $B1, B2 \leftarrow$ copies of the root node of $\pi$;
11     $\pi_{t1} \leftarrow$ **SplitTreesForWhile**($B1$, $D$, comp);
12     $\pi_{t2} \leftarrow$ **SplitTreesForWhile**($B2$, $D$, $\sim$ comp);
13     $\pi_{TAIG} \leftarrow$ A TAIG with two WHILE loops in sequence, the first WHILE
         loop having condition $W1$ and inner actions and conditions equivalent to
         $\pi_{t1}$, and the second WHILE loop having negated condition $W2$ and inner
         actions and conditions equivalent to $\pi_{t2}$
14 **end**
```

Part 1, creating the single-decision-tree style policy, can use the procedure from Section 7.4.2 to build a decision tree using reinforcement learning.

Part 2 utilizes the **FindSubtaskFeature** procedure, which is in Algorithm 8. If two sequential subtasks cannot be determined, then all return values are NULL and the procedure ends (no TAIG can be created). The procedure of **FindSubtaskFeature** observes the execution of a learned policy. Lines 1-4 simply describe the process of determining if there is one dimension that can be used to separate the task into two “stages”. If so, this is assigned to D. A boolean `comp` is returned that is True if the initial value of dimension D is lower than the final value, and False otherwise. Recall that in this constrained situation, a while-split will only be attempted if there are exactly two stages, so the initial and final values of this dimension are the only values it ever takes. Finally, a value u_D is returned as a midway point between these two values. It does not have to have any meaning, nor appear in the nodes of the policy π. Its purpose will be simply to differentiate between the initial and final values of dimension D.

Part 3 involves transforming policy π into a TAIG style policy with WHILE loops using the information obtained in part 2. Line 3 of Algorithm 7 checks whether such a transformation is possible. If **FindSubtaskFeature** returns all NULLs, this particular transformation is not possible and is not attempted. If values are returned, however, then the transformation proceeds, with lines 4 through 13 detailing the process. A TAIG will be created with two WHILE loops in sequence, each containing a series of If-Else Condition primitives and Action primitives. In other words, two decision trees are created, each of which are used to populate primitives inside the corresponding one of the two loops. In lines 4 through 9, the conditions for the condition primitive for each WHILE loop are generated. The first condition, $W1$, corresponds to the initial state of dimension D. The second condition, $W2$, is just "task is done" to ensure that the task will run to completion.

Algorithm 8: FindSubtaskFeature takes a policy π and determines if any features can be used to signify a change between two stages of a task.

```
1  FindSubtaskFeature($\pi$):
2  Allow observing an execution of policy $\pi$ to generate a vector of booleans $t_B$.
     This vector $t_B$ is of length equal to the length of $s$. Each element in $t_B$ is true
     if and only if the corresponding element in $s$ changed once and only once
     during task execution.
3  $T_B \leftarrow$ the union of all $t_B$ generated by executing policy $\pi$ $n$ times, for a large
     value of $n$;
4  if any($T_B$) then
5      $D \leftarrow$ the first element in $T_B$ that is true (the multi-stage signifying
         dimension);
6      $u_1 \leftarrow$ the initial value of of dimension $D$;
7      $u_2 \leftarrow$ the final value of dimension $D$;
8      comp $\leftarrow$ Boolean($u_1 < u_2$);
9      $u_D \leftarrow \frac{u_2 - u_1}{2}$ if comp else $\frac{u_1 - u_2}{2}$;
10     return $D$, comp, $u_D$;
11 else
12     The policy cannot be split into multiple stages.
13     return NULL, NULL, NULL;
14 end
```

In lines 10 through 12 of Algorithm 7, two decision tree policies are created from copies of π_0, according to **SplitTreesForWhile** (detailed in Algorithm 9). Policy π_{t1} is appropriate for executing in the first WHILE loop, and π_{t2} is appropriate for executing in the second.

Thus the final step in line 13 of Algorithm 7 is to take the two new decision tree policies and add them to the TAIG. Available Action Primitives correspond to the discrete set of actions in the action space. Conditional Primitives could be conceptualized in two ways: i) there is a single conditional primitive that take two parameters (dimension and value), returning true if the value of the dimension at the current observed state is less than value, and ii) there are a number of conditional primitives equal to the number of dimensions, each of which takes a single parameter (value), returning true if the value of the dimension at the current observed state is less than value. Branching nodes in the policies indicate how to form these Conditional Primitives for IF-ELSE structures, and Leaf Nodes indicate which Action Primitive to add. π_{t1} and π_{t2} by themselves could be represented as a decision tree policy or as a TAIG as described.

Algorithm 9: SplitTreesForWhile removes from the tree with root node at B any branches that do not satisfy the condition information.

```
SplitTreesForWhile($B$, $D$, comp):
if $B$ is leaf node then
    return $B$;
else
    if $B[m] = D$ then
        if comp then
            return $B$[left];
        else
            return $B$[right];
        end
    else
        $B$[left] ← SplitTreesForWhile($B$[left], $D$, comp);
        $B$[right] ← SplitTreesForWhile($B$[right], $D$, comp);
        return $B$;
    end
end
```

The **SplitTreesForWhile** procedure for obtaining π_{t1} or π_{t2} from π is shown in Algorithm 9. This procedure is a recursive process that progresses node by node through the initial policy. For `comp` == True, the resulting tree is intended to be the policy when dimension D is in its lower-valued state. Any node that splits on D ($B[m] = D$) has its left child returned. (Thus any node in π that splits on D will be replaced with whatever successors correspond to the lower-valued branch, and other child nodes discarded.) For `comp` == False, the reverse happens, since the higher-valued state is requested. Since D can only be two values (or else this procedure would not be executing), the procedure can take advantage of that fact to ignore the value u that B splits on in any case. It can use its u_D split value without detrimental effect. (If one were attempting to model a task with more than two stages, this would definitely not be a valid assumption, but here it is.) Leaf nodes are left unchanged. Branch nodes not splitting on D are unchanged themselves, but with children that are processed by **SplitTreesForWhile** in turn.

At the end, no nodes are left in π_{t1} or π_{t2} that split on dimension D, and each policy is suitable for either the lower-valued case of D or the higher-valued case.

Find a general illustration of the structure of of the result of Algorithm 7 in Figure 8.1.

In this manner, a policy can be learned autonomously through reinforcement learning and, if certain criteria are met, transformed into a TAIG-style policy.

8.3 Limitations

To use this approach with CQI or CQI2 (see Chapter 7), all of the requirements of the CQI or CQI2 methods must be satisfied. It would be feasible, however, to use the approach described in this chapter with any means of generating the pre-transformation decision tree policy, and then this limitation would not apply.

Another limitation is that the task to be learned must be in two clearly defined stages, signified by exactly one change-in-value for a dimension of the state, in order to obtain the while-loops.

A further pitfall occurs when there are two clearly defined stages, but the optimal policy is identical in each case. If the dimension that signifies the state change is not split upon in policy π ever, then π_{t1} and π_{t2} will be equivalent. While this should not affect performance, it does make the while-split unnecessary. This could be addressed by adding a check for the presence of such a node in π as an additional criteria before performing a split (if this were a priority).

Finally, a limitation of our method is that is cannot produce arbitrary TAIGs, or TAIGs with WHILE loops other than what is described here.

Even so, this approach is the first that generates a TAIG-style policy from a policy learned through RL.

8.4 Results

We test this technique on the Robot Navigation B environment.

The hyperparameters tested were all those that were noted as tested for Robot Navigation B environment for CQI2 in Section 7.5. Ten trials were run for each set of hyperparameters, where each trial consisted of two-million training steps (with $\epsilon = \max(0.05, 1 - \text{step}/d)$, where $d = 300,000$), evaluated with 50,000 steps before transformation, and evaluated with 50,000 steps after transformation (if

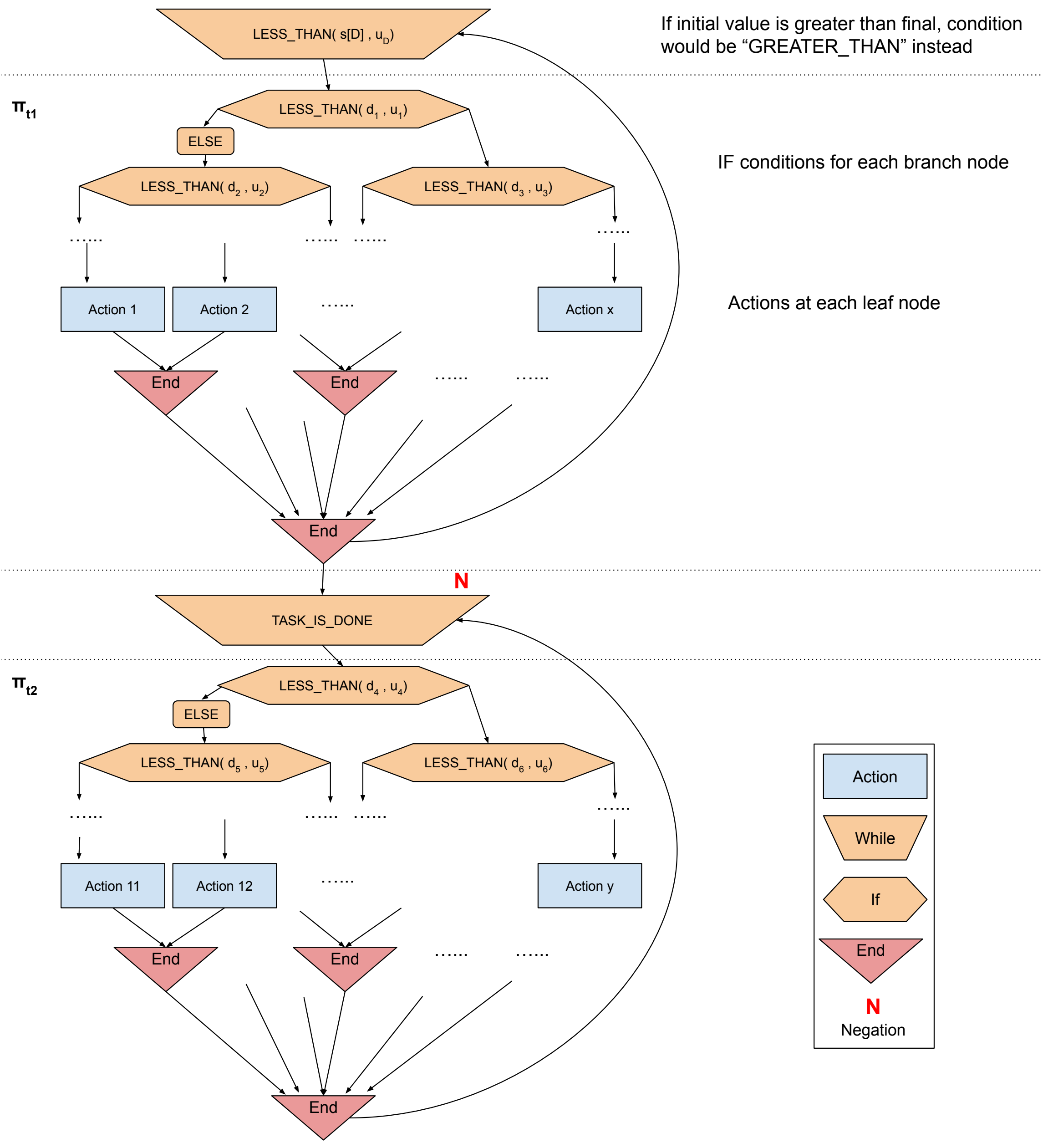

LESS_THAN(s[D] , u_D)
If initial value is greater than final, condition would be "GREATER_THAN" instead
π_{t1}
LESS_THAN(d_1 , u_1)
ELSE
LESS_THAN(d_2 , u_2)
LESS_THAN(d_3 , u_3)
IF conditions for each branch node
Action 1
Action 2
Action x
Actions at each leaf node
End
N
TASK_IS_DONE
π_{t2}
LESS_THAN(d_4 , u_4)
LESS_THAN(d_5 , u_5)
LESS_THAN(d_6 , u_6)
Action 11
Action 12
Action y
Action
While
If
Negation

Figure 8.1: Description of structure of TAIG policy for two-stage task after transformation from decision tree

				CQI2 Policy (π_0, Pre-Transformation)				**TAIG Policy**							
								π_{t1}		π_{t2}		π_{TAIG}			
α	H_S	n_S	% Trials Transform to TAIG	Avg. Size	S.D. Size	Avg. Reward	S.D. Reward	Avg. Size	S.D. Size	Avg. Size	S.D. Size	Avg. Size	S.D. Size	Avg. Reward	S.D. Reward
0.01	0.1	8	**100%**	148.8	35.60	-23.85	5.27	146.6	35.01	141.8	35.27	290.0	71.74	-23.43	5.04
0.001	1	2	90%	58.8	8.51	-19.83	1.02	58.0	8.12	48.4	11.93	108.4	16.13	**-20.16**	0.95
0.0005	1	9	90%	55.4	6.91	-28.65	14.25	55.0	7.55	52.3	11.66	**80.0**	18.68	-28.98	14.64
0.001	1	9	80%	42.3	15.71	-22.48	5.47	42.0	15.41	38.3	14.89	82.3	28.39	-22.72	4.94

Table 8.1: Best results for TAIG policy per %-successful-transformation group

transformation to TAIG occurred). Parameter 4 was the parameter that indicated the two stages of this task (when a transformation occurred, it meant that the algorithm correctly identified this piece of information).

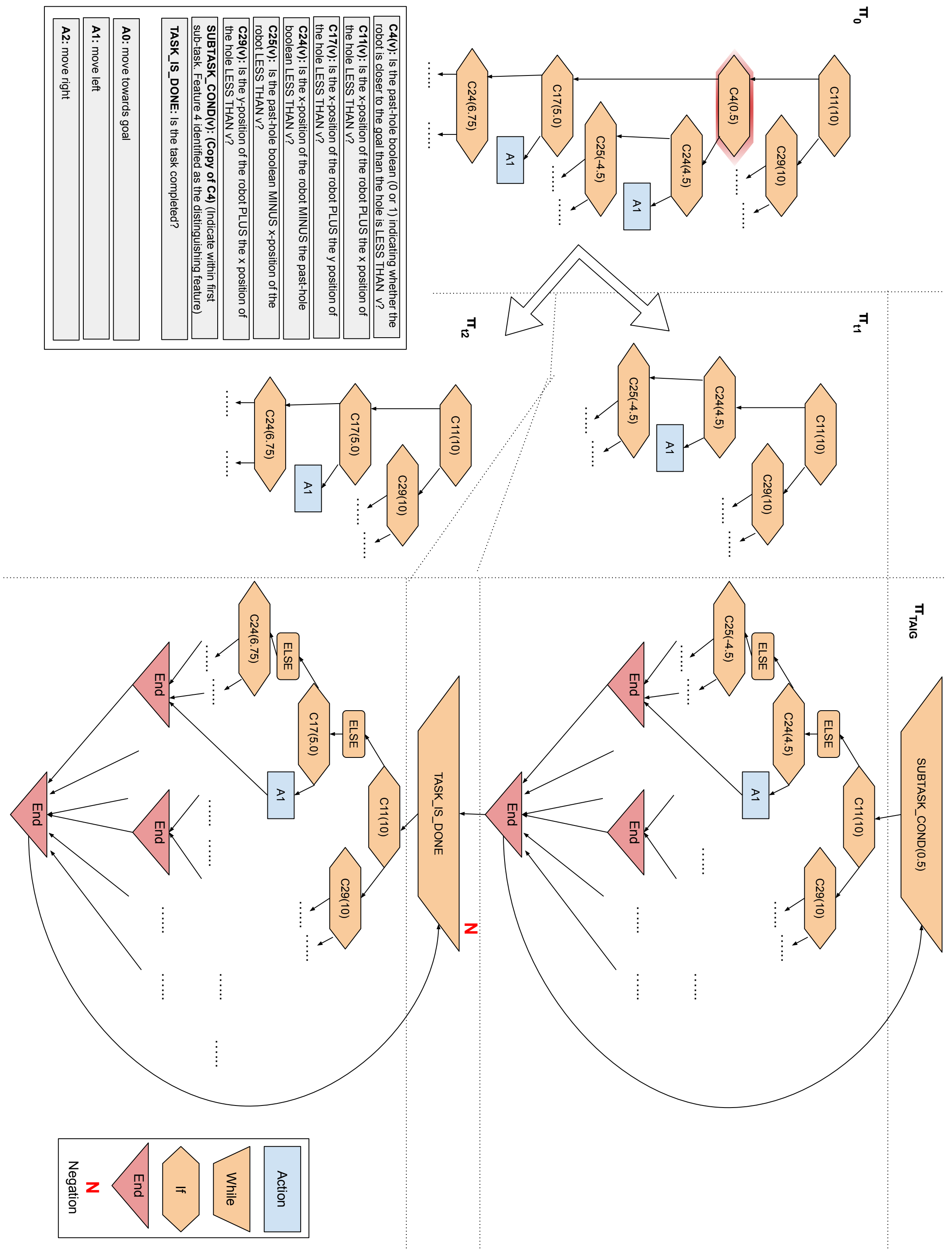
In Chapter 7, "reward achieved" and "size of policy" were valued metrics in a resulting policy. This chapter adds a third metric of value, which is the percentage of trials in which a policy is transformed into TAIG. (If there is no dimension that follows the two-stage pattern described in the previous section, the policy will not be transformed from its decision tree form.)

For example, with ($\alpha = 0.001, H_S = 1, n_S = 2$), this method achieves a transformation rate of 90%. These hyperparameters yield the highest reward among all policies where transformation-to-TAIG occurs 90% of the time. The reward of π_{TAIG} is -20.16. The average size of the decision tree π_{t1} is 58 and the average size of π_{t2} is 48.4. (The average size of π_0 is 58.8.) After transformation into a TAIG, the average total size of the TAIG (excluding ELSE, END-IF, and END-LOOP nodes) is 108.4. The trial for these parameters with the maximum total decrease in node size from π_0 to π_{t1} or π_{t2} was a transformation from π_0 of size 75 to π_{t1} of size 73 and π_{t2} of size 45.

Among the group of sets of hyperparameters that yield a successful TAIG transformation 100% of the time, the set of ($\alpha = 0.01, H_S = 0.1, n_S = 8$) results in both the highest average reward and smallest size. The reward of π_{TAIG} is -23.43. The average size of the decision tree π_{t1} is 146.6 and the average size of π_{t2} is 141.8. (The average size of π_0 is 148.8.) After transformation into a TAIG, the average total size of the TAIG (excluding ELSE, END-IF, and END-LOOP nodes) is 290. The trial for these parameters with the maximum total decrease in node size from π_0 to π_{t1} or π_{t2} was a transformation from π_0 of size 117 to π_{t1} of size 115 and π_{t2} of size 93.

See these results with additional information presented in Table 8.1. (Sizes of final TAIG are given excluding ELSE, END-IF, and END-LOOP nodes from the count.) In the headings, n_S refers to number of splits, H_S is the split-threshold-maximum, and "S.D." is an abbreviation for "Standard Deviation."

The entry for the 80% group in Table 8.1 is both the highest average reward and the smallest average size π_{TAIG} for the group.



C4(v): Is the past-hole boolean (0 or 1) indicating whether the robot is closer to the goal than the hole is LESS THAN v?

C11(v): Is the x-position of the robot PLUS the x position of the hole LESS THAN v?

C17(v): Is the x-position of the robot PLUS the y position of the hole LESS THAN v?

C24(v): Is the x-position of the robot MINUS the past-hole boolean LESS THAN v?

C25(v): Is the past-hole boolean MINUS x-position of the robot LESS THAN v?

C29(v): Is the y-position of the robot PLUS the x position of the hole LESS THAN v?

SUBTASK_COND(v): (Copy of C4) (Indicate within first sub-task. Feature 4 identified as the distinguishing feature)

TASK_IS_DONE: Is the task completed?

A0: move towards goal

A1: move left

A2: move right

Figure 8.2: Illustration of the TAIG transformation process for a subsection of a decision tree

The smallest average size π_{TAIG} across all hyperparameters tested across all groups regardless of %-success-transformation, and where the policy still achieves the task[1], is a size of 80, occurring with ($\alpha = 0.0005, H_S = 1, n_S = 9$). This policy has a post-transformation reward of -28.98, and falls into the 90%-success transformation group. It has an average maximum reduction from π_0 to either π_{t1} or π_{t2} of 24, and an average pre-transformation size of 57. The average reward pre-transformation was -29.06.

The parameters that yielded the largest average reduction in size from π_0 to either π_{t1} or π_{t2} (while still achieving the task), across all groups, are ($\alpha = 0.4, H_S = 1, n_S = 7$). Here, the average size of π_0 was 332.25 nodes, and the average size of π_{t1} and π_{t2} was 299.75 and 273.5, respectively, for an average maximum reduction of 58.75. With these parameters, the %-Transformation is 80%, and the average reward of the TAIG when a transformation does occur is -28.29.

Looking at the group of hyperparameters that achieve a transformation 100% of the time, slighter lower rewards and larger sizes are found. A user of this method thus may choose whether to prioritize ending up with a TAIG a larger percentage of the time, even if a worse overall policy results, verses successfully transforming into a TAIG less of the time but achieving a smaller size or higher-reward policy during those cases where it does transform. (Additionally, it is worth noting that the specific nature of this tradeoff may be domain dependent.)

Find an illustration of part of an example final policy in Figure 8.2. This is part of an example tree, with parameters ($\alpha = 0.01, H_S = 0.1, n_S = 5$). The π_0 has a total size of 89 nodes and achieves an average reward of -20.29. Feature 4 is identified as the subtask signifier—it is a boolean indicating whether the robot is past the hole yet or not (0 if the hole is closer to the goal than to the robot and 1 if the robot is closer to the hole than to the goal). The π_{t1} has 73 and π_{t2} has size 83. Shown in the diagram are sections near the top of the tree that change due to the transformation, followed by their completed transformation into TAIG form. The full size of the TAIG (excluding ELSE, END-IF, and END-LOOP nodes) is 158. The average reward it achieves is -21.07. Although the total policy is larger than π_0 and the reward shows a slight decrease, the TAIG form allows one to inspect differences in behavior and strategy that the robot employs during the course of the two different subtasks of the task.

The method discussed in this chapter allows for creating TAIG-style policies that can successfully solve the Robot Navigation B task. Average rewards pre- and post-transformation are very similar, well within a single standard deviation, demonstrating that the transformation to TAIG at optimal hyperparameters does not sacrifice efficacy. While the potential for trading off %-Transformation vs. Reward

[1]The term "achieving the task" in this instance is defined as "having an average reward above -30."

exists, it is also a testament to the soundness of the method that there exist parameters at which 100% transformation is possible while solving the task.

8.5 Summary

This chapter introduces a method whereby reinforcement learning is used to construct a TAIG. First, CQI2 is used to build a decision-tree style policy. Then a procedure is applied to transform the decision tree into a TAIG, with successive WHILE-loop clauses containing series of IF-ELSE clauses. This method also implicitly determines whether a feature distinguishes between two stages of a two-stage task.

The method presented here solves the Robot Navigation B task through autonomously learning a TAIG. It is capable of performing comparably with pCQI in some cases and outperforming the Pyeatt Method. Various tradeoffs that can be made when using this method to optimize for different metrics (reward, size, %-success-transformation) are discussed as well. This is the first time that a TAIG has been autonomously learned.

Thus, this chapter demonstrates the feasibility of using RL to learn a TAIG-style policy to solve a task.

Autonomously learning a decision-tree was valued in Chapter 7 due to the interpretability of such a structure. The contributions of this chapter involve enabling a robot to autonomously learn a TAIG, which is a difficult structure to learn. A TAIG has a much richer set of control flow structures than a decision tree, and corresponds to the structures created by humans in Chapters 3 through 5. When a robot learns a TAIG, it is learning a common, structured representation that enables interpretability of complex behavior.

Chapter 9

Conclusions

This chapter summarizes the work in terms of the contributions of the thesis and discusses future work.

9.1 Contributions

We have introduced TAIG, a system for task transfer that accounts for the disparities between robot systems that exist in the real world. We have provided an open source library for TAIG and a means of creating TAIG via interactive dialogue. We have introduced CQI and CQI2, extending pCQI. These are all methods for explainable reinforcement learning, allowing a robot agent to learn how to solve a task by developing a structured policy. Finally, we introduce a method for learning a certain type of TAIG with WHILE loops.

The TAIG paradigm adds negation to nodes and connects modular memory to the graph. These enhancements enable a robot to perform more complex tasks than they can with a simple IG. We have discussed the creation of the Primitive Library and the Memory-Library-Graph paradigm, by which functionalities can be transferred across tasks and across robots. We demonstrated that with TAIG, whole tasks in and of themselves can be directly transferred between robots without any modification of the graph. Using the paradigm introduced in this paper also reduces development time since primitives and Instruction Graphs can be re-used.

We discussed how nesting TAIGs inside action primitives in other TAIGs and how the use of halt conditions can be used to build structures for performing complex tasks with subtasks. We demonstrated this with a complex GPSR task.

Using TAIG, a single task plan can be specified for robots with wildly different hardware and technical capabilities, and a single robot can understand new tasks by building on existing knowledge.

We demonstrated how the interactive dialogue system can be used to give instructions to a robot to build up its knowledge of how to perform a task, using previously known primitives or graphs.

We noted the release of the `instruction_graph` library, which implements TAIG. The library includes capabilities for building the components of the interactive dialogue, as well as every other capability discussed or utilized in Chapters 3 through 5. The robotics community may find it useful for accelerating the development of robotic systems.

We introduced Conservative Q-Improvement and Conservative Q-Improvement 2, extensions of partial-CQI which enable a robot agent to develop a structured representation of a task policy in the form of a decision tree via reinforcement learning. CQI2 can be used in any environment with discrete actions and a multidimensional state space. It produces a policy in the form of a decision tree, and, like the pCQI method before it, produces smaller trees than existing methods while not sacrificing policy performance. We investigated the nature of the CQI2 method, discussing ways that one might optimize for tradeoffs between smaller trees and strictly better policies. We found that pCQI out-performed the Pyeatt method and that CQI outperformed pCQI. This is due to CQI's conservative nature, whereby it only creates a new node when doing so will result in an improvement in the policy.

CQI2 builds on CQI by expanding the state space, building new features out of combinations of existing features. CQI2 outperforms CQI in some cases, since certain relationships can be taken advantage of using a single node split that previously required multiple nodes. One caveat is that due to the expanded state space, CQI2 takes longer to train than pCQI or CQI.

The tradeoffs of CQI/CQI2 can be tuned depending on the use case, and depending on the domain. For example, if there were a domain where CQI/CQI2 produces large trees, a practitioner could settle for lower reward in order to have a smaller tree that can be understood and inspected. On the other hand, if all policies for an environment were relatively small, or if the policy only had to be manually inspected occasionally or without hurry, then one could choose parameter values which result in a larger, better-performing policy.

Finally, we incorporated learning decision-tree policies into a methodology for learning a TAIG. This methodology can be used to solve a task that i) has exactly two stages, delineated by a value change in a single feature of the state space, and ii) can be solved with a decision tree policy alone. The methodology results in a TAIG, where two WHILE loops are arranged in sequence with a (potentially) large series of IF-ELSE clauses inside each. We demonstrated how this transformation can be performed resulting in a TAIG that is learned and developed completely autonomously yet is able to solve a task satisfying the criteria.

Structured, interpretable policies such as decision trees or TAIGs have many benefits, whether constructed or learned. If one developer creates a policy for a robot, a different developer can see what the robot will do without having to ask the human, just by looking at the structure. When a robot learns a policy, a human does not have to wonder why a robot is taking an action or what the robot will do in a specific scenario—the policy is interpretable for all to see. Another example is in a scenario where an end-user wishes to continually tweak a policy, or change it after it has been learned.

Autonomous robots will be everpresent in the world of the future, but they need not be a mystery. Structured representations for behaviors of autonomous robots may allow robots to serve humanity more successfully, and to be better understood by engineers and end-users alike.

9.2 Future Work

Possible future work for TAIG could include adding validation checking, to ensure runtime compatibility between a Memory Object, a Primitive Library and a Graph. This could be a software script or an add-on to an Interactive Development Environment (IDE). Currently it is up to the developer or robot operator to ensure that this compatibility exists.

Future work can expand the Interactive TAIG to be even more useful, such as adding the ability to modify the graph in place (as opposed to overwriting an existing graph in full if a change is desired). There are also opportunities to create more complex requests. For example, “change all tuples of type x to have parameter y”, along with the parsing necessary to support that (it may entail going beyond regular expressions, using a more sophisticated NLP technique).

Another means of interactively consuming and creating a TAIG would be through a graphical display. The TAIG could be displayed in a GUI and the user could drag tuples around to change the order, or add tuples from the library in another pane.

Future work regarding CQI or CQI2 could entail modifying the algorithm to utilize tree-building methods that are not strictly additive but rebalance the tree along the way. This could result in a further reduction of final policy size. Additionally, a regularization term which allows the tree size to directly affect the policy's perceived reward would allow CQI to directly optimize for a trade-off between size and performance on original tasks. Since interpretability is one of the goals, a user study could be performed to quantify this aspect of the method.

Future work for learning a TAIG with While Loops could involve developing technique similar to that presented in this thesis, but expanding Algorithms 7 through 9 to handle additional numbers of subtasks (three or more). The criteria would be some degree of feature stability for periods of time. This would allow creating longer sequences of while loops representing stages of a task.

There is also much future work to be done on learning while loops in other contexts.

Other types of TAIG-compatible structures are also worth learning, such as sequences of actions. Future work could also entail developing additional methods of learning tasks that take advantage of TAIG's other capabilities, such as halt conditions.

Further opportunities for future work could be discerned by pondering the goals and ideas about the future expressed by the direction taken by this thesis, and considering the next step with regards to the work that remains.

One of the main visions for this thesis is robots learning to solve tasks through a combination of inputs (self-learned, programmed/instructed, and collaboration with a human) instead of only one. Beyond the learning of TAIGs, there is worthwhile work to be done toward realizing this vision, where over time humans and robots build a shared understanding of the world and how to achieve our goals, each one building on the knowledge and strengths of the other.

This thesis helps to bridge the gap between constructed or instructed policies and learned policies. The above suggestions describe the path forward to bring about a better world for tomorrow.

Appendix A

Developer's Tutorial for Using TAIG With Pepper

In this Appendix, please find an introductory tutorial for using the TAIG library with a Pepper robot. For an explanation of the TAIG paradigm, see Chapter 3.

A.1 Using TAIG for a Task

This section demonstrates how to use TAIG. We are going to create a simple task for the Softbank Pepper robot.[1] The task we will create is for Pepper to naively search for a human by spinning around, and to greet them when it finds one. This task graph and the primitive library for this task are shown in Figure A.1.

Install TAIG as a Python library with pip[2]:

[1]For those without access to a Pepper, more trivial examples that do not require a robot of any kind are available in the library's GitHub repository documentation.

[2]Find it on pypi: https://pypi.python.org/pypi/instruction-graph, or view the source code at: https://github.com/AMR-/instruction_graph

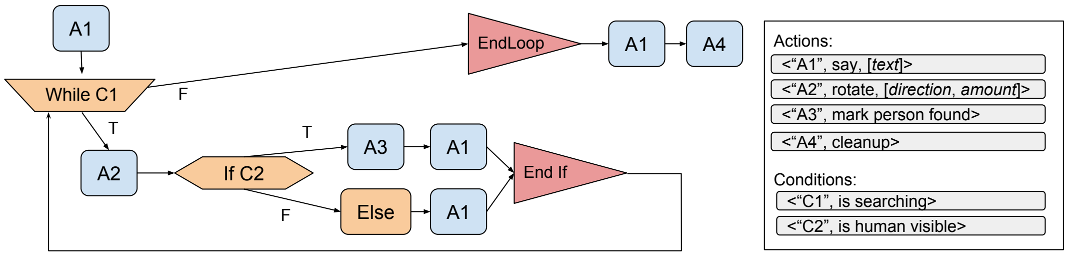


Figure A.1: An example of the structure of the Task Graph and Primitive Library.

```
$ pip install instruction_graph
```

We will create attributes in the Memory Object to store state information, Pepper's session information, and variables we want to track. We then create a series of primitives to execute the component action and condition checks. Finally we write a controller that builds the Graph and saves it to a file, and that can also load and run the graph.

A.1.1 The Memory Object

The Memory Object is where we store information we want to track over time. Since we are using Pepper, we need to track the naoqi session. We also have two states, "SEARCHING" and "FOUND_PERSON," among other information.

Download the example file PepperMemory.py[3]. This is the file where the Memory Object is created.

The Memory Object class should extend BaseMemory. It can declare whatever attributes it needs to in *__init__*. It can have helper functions, as in *cleanup* here. Every Memory Object must implement *memory_name*, which returns a descriptive string (used in logging).

Note that if the project uses ROS, subscribers to nodes and publishers should be set up in the Memory Object.

A.1.2 The Primitive Library

Next we consider the atomic units of work and condition checks that make up our task. At this stage, if we have previously implemented tasks (on this or other robots), we might want to look at our existing Primitive Libraries to see what can be re-used. Perhaps one of our existing libraries has everything we need. If it is our first TAIG, of course, we will have to create a new Primitive Library.

For our simple task we want four Action Primitives ("say", "rotate", "person_found", and "cleanup") and two Conditional Primitives ("is_searching" and "is_human_visible").

We will create a **PepperPrimitiveLibrary.py** file. The full example file can be downloaded from the repository[4]. The Primitive Library class should extend BasePrimitiveLibrary.

A single primitive is essentially a function with metadata. First, define the function. In the excerpts in Figure A.2, we see two example methods. These methods will later be referenced by the "say" Action Primitive and "is_pepper_searching" Condition

[3]`https://github.com/AMR-/instruction_graph/blob/v0.2.8/instruction_graph/example/PepperMemory.py`

[4]`https://github.com/AMR-/instruction_graph/blob/v0.2.8/instruction_graph/example/PepperPrimitiveLibrary.py`

```
  def say(memory, text):
      tts = memory.session.service("ALTextToSpeech")
      tts.say(text)

  def is_pepper_searching(memory):
      return memory.state == States.SEARCHING
```

Figure A.2: Functions for use in Primitives

```
ActionPrimitive("say", self.say,
                "Say", "Do text-to-speech on input argument")
ConditionalPrimitive("is_searching", self.is_pepper_searching)
```

Figure A.3: ActionPrimitive and ConditionalPrimitive

Primitive. Note how both are stateless. The Memory Object will be passed as the first argument when the function is called, so every primitive function should include it even if the function does not utilize it.

There are three required functions.

- *list_action_primitives* - return a list of ActionPrimitives. ActionPrimitives and ConditionalPrimitives are instantiated as shown in Figure A.3. The arguments are i) *pid* string, ii) the function itself, iii) & iv) optional human-readable name and description. (There are additional optional arguments for advanced use not covered in this tutorial.)
- *list_conditional_primitives* - return a list of ConditionalPrimitives
- *library_name* - return a string that is a descriptive name for the library

A.1.3 Controller: Graph Creation and Execution

We can create the graph inside whatever framework we desire. In our sample application here we are going to create a controller object that will create and run the graph. We will create a **PepperController.py** file. The full example file can be downloaded from the repository[5].

We use the Manager to create, save, load, and execute graphs. When we instantiate a Manager we specify the particular Primitive Library and Memory Object to

[5]https://github.com/AMR-/instruction_graph/blob/v0.2.8/instruction_graph/example/PepperController.py

use. Here the Manager is initialized with instances of PepperMemory and PepperPrimitiveLibrary. Any graphs executed with this manager will use the associated Primitive Library and Memory Object. A graph created with this Manager can be saved to file and used with any compatible Primitive Library or Memory Object, not just that with which it was created.

The *build_instruction_graph* method shows how to use a manager to create a new graph and save it. (It builds the graph shown in Figure A.1.) First it calls *create_new_ig* on the Manager. Now there is a Graph on the Manager's ig attribute. Nodes of various type (actions, if, loops) can be added. When specifying an action or condition, include as the first argument the Primitive's *pid*, and if necessary include a list of arguments in *args*. Note that we do **not** explicitly pass the Memory Object, so the list of arguments we pass should be one less than the number of arguments in the function referenced by the referenced Primitive. Refer to the example file for examples for various kinds of nodes. After the graph is created, we can save it to file by calling *save_ig* on the Manager and passing the filename.

The *run_instruction_graph* method loads the task graph file with *load_ig* and then runs it with *run*.

A.1.4 Let's Run It

Download the **pepper_build.py** example script[6] (creates the graph and saves to a file named **pepper_demo.ig**) and the pepper_run.py example script[7] (executes an existing graph file named **pepper_demo.ig**). Note that they are exactly the same except for the last line. Run them as follows:

```
$ python pepper_build.py
  --ip [IP.TO.YOUR.PEPPER] --port 9559
$ python pepper_run.py
  --ip [IP.TO.YOUR.PEPPER] --port 9559
```

Figure A.4: Pepper using TAIG

After a task graph has been created once, the same file can be re-used. See a picture of our Pepper taken while searching for a human in Figure A.4.

[6] https://github.com/AMR-/instruction_graph/blob/v0.2.8/instruction_graph/example/pepper_build.py

[7] https://github.com/AMR-/instruction_graph/blob/v0.2.8/instruction_graph/example/pepper_run.py

A.2 Conclusion

We introduce TAIG as a task definition framework in which we construct high level tasks out of stateless behavioral and perceptual primitives with access to memory. The TAIG library can help a robotics team accelerate the development of new and existing systems by re-using existing functionality. TAIG encourages clean organization wherever it is used in the codebase, a quality that helps a robot (or any complex system) stand the test of time.

We contribute this open-source library and corresponding pip package and make it available for use by the robotics community. It is well-tested[8] and development continues apace. We ourselves have used TAIG at Carnegie Mellon University to streamline the development of new tasks on Pepper as well as transfer tasks between disparate robot systems (Baxter and Pepper).

[8]Tested on python 2.7 and 3.5, with tests included in the repository.

Appendix B

Documentation for TAIG Open Source Library

In this section find the documentation that comes with the TAIG open source library software.

Transferable Augmented Instruction Graph (TAIG) is a library that allows the creation of task plans for robots or for other agent-systems. These task plans can be transferred across systems very easily.

Tasks are created in a graph form with conditionals and loops. Nodes in the graph refer to “primitives” which are atomic units of work (actions for the system to perform) or conditions to test.

Create an Instruction Graph, and associate it with a Primitive Library and Memory Object (noted below) and you can execute the task on a system.

This paradigm is useful because it allows executing a single task plan across multiple robots/systems. For a single robot, allows defining atomic functionality once, and re-using it across all the tasks that that robot is to complete.

Table of Contents

1. Installation
2. Introduction
3. QuickStart / Example
4. Details for Developing Your Own TAIG
    1. Creating Memory
    2. Creating Primitives
    3. Creating the Graph and Saving to File
    4. Running the Graph
    5. Additional Techniques
    6. Nested Graphs

5. Interactive Taig
6. Credits

B.1 Installation

To install the library just run

```
pip install instruction_graph
```

instruction_graph has been tested with Python 2.7 and 3.6.

B.2 Introduction

There are three components to the paradigm:

1. The Memory Object
2. The Primitive Library
3. The Instruction Graph

The Memory Object has fields that store any information required by the application at runtime (session info, database connection, ROS topics, state information, and any other data the application will track and store).

The Primitive Library is an object which holds a collection of Primitives. A Primitive can be either an Action or a Condition. Actions are simple actions that are performed. Conditions are simple conditions that are tested, and can be used in an IF or WHILE node. Each Primitive has at least a Primitive ID and a function. (See more details below.)

The Instruction Graph is a directed graph. Each node contains a reference to a primitive. When the graph is traversed, the function held by the Primitive to which the node refers is executed. (See more details below.)

The Memory Object provides the memory, and should contain no task logic. The Primitive Library contains Primitives with atomic functionality. Primitives should be divided into robot/system-specific Primitives and task-specific Primitives (this organizational division is not required, but is just for your own benefit). Primitives should not have any task logic nor should they refer to system memory directly, but rather should use the Memory Object to read/write any data they require. Primitives should be stateless. The Instruction Graph contains the task logic.

These three components are modular. You can switch one out without touching the other two.

During graph creation or execution, we say that a graph is “associated” with a Primitive Library and Memory Object. This association is performed by Manager.py.

B.3 QuickStart / Example

To just quickly run an instruction graph, you can use the example Memory and Library that ships with instruction_graph.

You can run and execute the following code:

```
from instruction_graph import Manager
from instruction_graph import DefaultMemory,
 ↪  ExamplePrimitiveLibrary

COUNT = "count"
HOW_COOL = "how cool is TAIG?"

A_SET = "fun_set"
A_GET = "fun_get"
A_PRINT = "print_args"
C_LESS = "less"

mem = DefaultMemory()
lib = ExamplePrimitiveLibrary()
m = Manager(memory=mem, library=lib)

m.create_new_ig()
m.ig.add_action(A_SET, args=[COUNT,5])
m.ig.add_action(A_SET, args=[HOW_COOL,"So awesome and cool."])

m.ig.add_if(C_LESS, args=[COUNT, 10])
m.ig.add_action(A_PRINT, args=["The count is less than %d", 10])
m.ig.add_else()
m.ig.add_action(A_PRINT, args=["The count is NOT less than %d",
 ↪  10])
m.ig.add_end_if()

m.ig.add_action(A_GET, args=[HOW_COOL])
m.save_ig("graph.ig")

m.load_ig("graph.ig")
m.run()
```

You are creating a graph using the example Memory and Primitive Library. It will set two values in the memory, check one of them in an if condition and print, and then get the value of the other and print it.

This is a basic example, of course, so that you can understand it easily.

B.4 Details for developing on your own TAIG

Memory, Primitives, and Graph are decoupled and modular. You can use different Memory Objects with the same or different Graphs, and different Primitive Libraries with the same or different Graphs. So, you can create them and combine them in any order.

B.4.1 Creating Memory

Memory is a good place to start as you create your own system. Technically Memory is not required. If your application is totally reactive and stateless, then you can just set the memory to `None` in the Manager.

Probably you will want some memory though.

When creating the memory object, consider all the types of information that you may want to store. This could be containers for application state information, connections to databases, or anything else you will need.

Create a Python file, for example `example_create.py`

Consider this class, similar to DefaultMemory in the QuickStart example:

```python
from instruction_graph.components.Memory import BaseMemory

class DefaultMemory2(BaseMemory):
    def __init__(self):
        super(DefaultMemory2, self).__init__()
        self.info = {}
        self.database_connection = None
        self.counter = 0
        self.whatever = "data"

    def memory_name(self):
        return "Another_Example_Memory"
```

It has attributes that an application can use. A Memory object can define any attributes.

Memory Object should extend BaseMemory, and implement the `memory_name` method.

If you want your application to publish to a ROS topic, we recommend adding the `rospy.Publisher` object as a value to an attribute of the Memory object. If you want your application to subscribe to a ROS topic, we recommend adding that to the

Memory as well, along with any callback (the callback could update additional values in the memory). Primitives should not subscribe to topics directly, and Primitives should publish to ROS topics by referencing the Publisher on the Memory object.

B.4.2 Creating Primitives

Primitives are where the actual low-level functionality for executing task components is stored.

There are two kinds of Primitives, Actions and Conditions. Actions store atomic functionality, and are meant to be used on Action nodes. Conditions check conditions, and are meant to be used on IF or WHILE nodes.

Primitive functions can be parameterized (they can take arguments).

To create a primitive, you will first define a function. A function meant for an action primitive should not have a return value. A function meant for a condition primitive should return `True` or `False`.

See an example Primitive Library defined below.

```
from instruction_graph.components.PrimitiveLibrary import
↪ BasePrimitiveLibrary
from instruction_graph.components.PrimitiveTuples import
↪ ActionPrimitive as Action, ConditionalPrimitive as Cond


class ExamplePrimitiveLibrary2(BasePrimitiveLibrary):
    def library_name(self):
        return "Example_Primitive_Library_2"

    def list_action_primitives(self):
        return [
            Action(fn_name='set', fn=self.set_value, human_name='Set
             ↪ Function', human_description='Sets a value in the
             ↪ memory.'),
            Action("print_args", self.print_args, "Print with Args",
             ↪ "Print the first argument interpolated with the
             ↪ second."),
            Action("dec", self.decrement, "Decrement Key",
             ↪ "Decrement the value found at the specified key by
             ↪ 1")
        ]

    def list_conditional_primitives(self):
        return [
            Cond('less', self.check_if_less, human_name='is less',
             ↪   human_description="Checks if the value of a certain
             ↪   key is less than a given value. (Returns true if
             ↪   so.)")
        ]

    # Actions #

    @staticmethod
    def print_args(memory, text, args):
        print(text % args)

    @staticmethod
    def set_value(memory, key, value):
        memory.info[key] = value
        print('%s set to %s' % (key, value))

    @staticmethod
    def decrement(memory, key):
        value = memory.info[key] - 1
        memory.info[key] = value
        print("%s: %s (decremented)" % (key, value))

    # Conditions #

    @staticmethod
    def check_if_less(memory, key, maximum):
        value = int(memory.info[key])
        return value < maximum
```

Note how functions are defined and then referenced in the ActionPrimitive and ConditionPrimitive instantiations.

Required methods are:

- library_name - a string to indicate this library's name, used in logging
- list_action_primitives - should return a list of ActionPrimitives
- list_conditional_primitives - should return a list of ConditionalPrimitives

B.4.3 Creating the Graph and Saving to File

Let's use our Memory and PrimitiveLibrary from above in creating an instruction graph.

We instantiate a Manager object, specifying the Memory and PrimitiveLibrary objects that we created above.

This particular graph will set the value of "count" in the memory to 6.Then it will kick off a loop that will run until "count" is less than 1. In each iteration of the loop it will check whether "count" is less than 3. If so, it will print "count is less than 3" and if not it will print "count is greater than or equal to 3". Then "count" will be decremented.

Finally, the graph will be saved to "graph_filename.ig"

```
from instruction_graph import Manager

mem_obj = DefaultMemory2()
eg_library = ExamplePrimitiveLibrary2()
igm = Manager(library=eg_library, memory=mem_obj)
ct = "count"
igm.create_new_ig()
igm.ig.add_action("set", args=[ct, 6])
igm.ig.add_loop('less', args=[ct, 1], negation=True)

igm.ig.add_if('less', args=[ct, 3])
igm.ig.add_action("print_args", args=["count is less than %d", 3])
igm.ig.add_else()
igm.ig.add_action("print_args", args=["count is greater than or
↪  equal to %d", 3])
igm.ig.add_end_if()

igm.ig.add_action("dec", args=[ct])
igm.ig.add_end_loop()
igm.save_ig("graph_filename.ig")
```

Run this code to check it out!

B.4.4 Running the Graph

After you have created "graph_filename.ig," you can load it and run it.

Use the following code to do so (you can reuse the existing Manager or create a new one as show). This can be in the same file, or in a new file called `example_run.py`

```
from instruction_graph import Manager
from example_create import DefaultMemory2,ExamplePrimitiveLibrary2


mem_obj = DefaultMemory2()
eg_library = ExamplePrimitiveLibrary2()
igm = Manager(library=eg_library, memory=mem_obj)

igm.load_ig("graph_filename.ig")
igm.run()
```

Note that the graph can be run on a system right from the file. You do not need to create anew. Make sure that the Primitive Library and Memory Object you use with a graph are compatible.

B.4.5 Additional Techniques

There is a special type of node you can add to a graph called a 'halt condition'. Whenever the halt condition becomes true, the graph execution immediately stops. To add a halt condition, use the *set_halt_condition* command. Add a line according to the following example anytime during graph creation:

```
igm.ig.set_halt_condition("less_or_no_key", args=["key1", 7],
 ↪  negation=True)
```

In this case, the example primitive used with the arguments shown in the example line mean that the graph execution would halt if *key1* was defined and if its value exceeded 7. Like other conditional primitives, this can be negated or not, and used with whatever arguments you desire.

This is useful if you are on a robot system with an emergency stop button, or if you want your agent to terminate execution immediately under other scenarios.

B.4.6 Nested Graphs (Graphs as Primitives)

You can use the special built in "run_ig" action primitive to include a node that attempts to run a different graph. In this manner, graphs can be nested within other graphs. The manager will attempt to look for the graph in the same directory to which all graphs are being saved (so if you are going to used previously saved graphs, ensure these directories are the same).

When running a sub-graph, the entire graph executes before the node on the upper graph is exited. The library and memory in the parent graph are passed to the child during execution, and updates to the memory object made by the child graph execution will persist when execution continues on the parent graph. There is no limit to how much you can nest graphs (aside from the practical consideration of computer memory).

The default name of the primitive to run a graph is "run_ig". If this is the name of another primitive in the user-defined library, a different name is chosen. If the library subclasses BasePrimitiveLibrary as described in this documentation, then the name of the primitive will be stored in the **run_ig_name** attribute of the library, and can be retrieved therefrom.

There are two ways to run a graph: explicitly by name and by checking the queued name. (The latter allows for the dynamic selection of a graph to run.)

To run a sub-graph explicitly, consider the following line:

```
igm.ig.add_action(igm.library.run_ig_name,
 ↪  args=["path-to-graph-as-string"])
```

This will put a node in the graph that contains the instruction to run the graph *path-to-graph-as-string* when it is executed.

Please do note that it is the instruction that is added to the graph and not the child graph itself. If the subgraph changes (if a new file is saved to the same filename), it is this new graph that will be loaded and executed at runtime.

To run a sub-graph dynamically, find the structure to do so in the following two lines:

```
igm.ig.add_action("queue_ig", args=["path-to-graph-as-string"])
igm.ig.add_action(igm.library.run_ig_name)
```

A path to the graph to run is queued up with the "queue_ig" primitive. Then, *run_ig* is run without arguments to indicate accessing it this variable .

The "queue_ig" method is not included by default, but it is in the example primitive library where it is defined as follows:

```
    @staticmethod
    def queue_ig_prim(memory, path):
        memory.queue_ig_as_primitive(path)
```

You can copy this implementation or write your own to achieve this functionality. If you use a Memory that extends BaseMemory, then the **queue_ig_as_primitive** and **get_queued_ig** methods will be inherited by the memory and can be referenced in the above manner.

If a halt condition is included on a graph used as a child graph, and the halt condition becomes satisfied, child graph execution will terminate, and execution of the parent graph will continue on to the next node. This has two ramifications:

1. The parent graph does not know if the child graph completed successfully
2. Sub-tasks can be terminated without stopping agent execution

B.5 Interactive TAIG

The Interactive Manager (IM) extends the Manager and acts as an agent. You, another human, or another computer system can then interact with the IM over a text-based interface to execute primitives, create task graphs, and executive task graphs.

There are a few steps towards using the IM.

First, define Primitives with appropriate elements. Then instantiate the IM and use it.

When the IM is evaluating an input, it attempts to make sense of the request, and handle it, and respond.

B.5.1 Primitive Library for IM

Defining a Primitive Library for use with an IM is similar to defining it for use normally. Primitive functions are implemented in the same way. The only difference is that when creating *ActionPrimitive* and *ConditionPrimitive* objects and listing them in the **list_action_primitives** and **list_conditional_primitives** functions, additional properties are added.

See an example Primitive Library defined below.

```
from instruction_graph.components.PrimitiveLibrary import
 ↪  BasePrimitiveLibrary
from instruction_graph.components.PrimitiveTuples import
 ↪  ActionPrimitive as Action, ConditionalPrimitive as Cond


class ExamplePrimitiveLibrary2(BasePrimitiveLibrary):
    def library_name(self):
        return "Example_Primitive_Library_2"

    def list_action_primitives(self):
        return [
            Action(fn_name='im_set', fn=self.f_set, human_name='Set
             ↪  Function',
                   human_description='Sets a value in the memory.',
                   match_re_or_fn="set [a-z0-9 ]+ to [a-z0-9 ]+",
                    ↪  argparse_re_or_fn="set (.*) to (.*)",
                   parsed_description=lambda args: "Set the value of
                    ↪  '%s' to '%s'" % (args[0], args[1])),
            Action(fn_name='im_inc', fn=self.increment,
             ↪  human_name='Increment Key',
                   human_description='Increment the value found at
                    ↪  the specified key by 1.',
                   match_re_or_fn="inc(rement)? [a-z0-9 ]+",
                    ↪  argparse_re_or_fn="inc(?:rement)? ([a-z0-9
                    ↪  ]+)"),
            Action(fn_name='im_dec', fn=self.decrement,
             ↪  human_name='Decrement Key',
                   human_description='Decrement the value found at
                    ↪  the specified key by 1.',
                   match_re_or_fn="dec(rement)? [a-z0-9 ]+",
                    ↪  argparse_re_or_fn="dec(?:rement)? ([a-z0-9
                    ↪  ]+)",
                   parsed_description=lambda args: "Decrement %s by
                    ↪  one" % args[0]),
        ]

    def list_conditional_primitives(self):
        return [
            Cond(fn_name='less', fn=self.check_if_less,
             ↪  human_name='is less',
                 human_description="Checks if the value of a certain
                  ↪  key is less than a given value.  "
                                   "(Returns true if so.)",
                 match_re_or_fn="[a-z0-9 ]+ is less than [0-9]+",
                 argparse_re_or_fn="([a-z0-9 ]+) is less than
                  ↪  ([0-9]+)",
                 parsed_description=lambda args: "The value of '%s'
                  ↪  is less than %s" % (args[0], args[1])),
        ]

    # Actions #

    @staticmethod
    def f_set(memory, key, value):
        memory.set(key, value)
        print('%s set to %s' % (key, value))

    @staticmethod
    def increment(memory, key):
        value = int(memory.get(key)) + 1
        memory.set(key, value)
        print("%s: %s (incremented)" % (key, value))

    @staticmethod
    def decrement(memory, key):
        value = int(memory.get(key)) - 1
        memory.set(key, value)
        print("%s: %s (decremented)" % (key, value))

    # Conditions #

    @staticmethod
    def check_if_less(memory, key, maximum):
        value = int(memory.info[key])
        return value < maximum
```

This is very similar to the example Primitive Library noted before, with additional Primitive attributes as follows:

- **match_re_or_fn**: This is a regular expression or function (returning a boolean). When the IM receives text input, it uses this parameter to determine if this is a primitive to which the text is referring. If the regex matches, or the function returns true, it is considered a match.
- **argparse_re_or_fn**: This is a regular expression or function (returning a list of strings). After a piece of input text has been identified as referring to a primitive, this function is called to determine how to parameterize it. If it is a regular expression, each capturing group is a parameter. If **argparse_re_or_fn** is omitted, no parameters are passed to the primitive.
- **parsed_description**: This is a function that takes in a list of arguments and returns a string. It is used by the agent to respond to the user, for describing a parameterized primitive. If **parsed_description** is not specified, the **human_description** is used.

Note that the primitives you create for use in the interactive manager might receive their arguments as strings. You should ensure that the functions defined for the primitives themselves do any type conversion required.

B.5.2 Using the Interactive Manager

Find an example of instantiating the Interactive Manager with example Memory and Primitive Library below.

```
from instruction_graph.interactive.InteractiveManager import
 ↪  InteractiveManager
from instruction_graph.example.DefaultMemory import DefaultMemory
from instruction_graph.example.ExamplePrimitiveLibrary import
 ↪  ExamplePrimitiveLibrary

memory_obj = DefaultMemory()
library = ExamplePrimitiveLibrary()
im = InteractiveManager(library=library, memory=memory_obj)

resp = im.parse_input_text("set key 1 to 3")
print(resp)
resp = im.parse_input_text("I will teach you to dance")
print(resp)
...
resp = im.parse_input_text("Done Learning")
print(resp)
```

The IM is initialized with a library and a memory. It creates a Manager inside itself which it uses to build a graph if necessary.

The **parse_input_text** method takes in text. The IM agent will perform any actions and return its response as a string. Actions may involve executing primitives/graphs or building/saving graphs.

The IM can exist in four states:

- WAITING: can be commanded to execute a primitive, execute a graph, or start learning a new graph
- CONFIRM_LEARN_IG: after being asked to start a new graph, the IM will ask for confirmation
- LEARNING_IG_WAITING: During construction of a new graph, can be instructed to add a primitive to the graph, or stop learning the graph
- CONFIRM_ADD_PRIM_WHEN_LEARNING: when the IM believes it has heard a primitive to add to the graph, it will repeat what it has heard and ask for confirmation. The state is stored on the state attribute (*im.state*) in above. It should never be necessary to access, it is described here merely of interest and as it may help you understand what is going on.

The manner in which a primitive is recognized is the same for executing or adding. **match_re_or_fn** is used to match, **argparse_re_or_fn** is used to parse primitives, and **parsed_description** is used during the confirmation request. Matching attempts are processed in the order in which they are listed in the return from **list_conditional_primitives** or **list_action_primitives**.

One primitive is added to the graph at a time.

The grammar for the communication itself is noted by the Builder Phrases, noted in the next section.

B.5.3 Builder Phrases

Please find in Table B.1 the Builder Phrases, which entail the commands or requests that the IM will understand. The first column shows the attribute name in the BuilderPhrases object, the second column shows the (configurable, default) value, and the third column has a description of the purpose and meaning. In the table, the non-IM agent is referred to as a human for clarity, although, as mentioned, it doesn't have to be a human, it could be another computer agent.

Any text input not matching a phrase from Builder Phrases is assumed to refer to an action primitive that should be executed or added.

You can customize your own BuilderPhrases and use those with the Interactive Manager, as shown below.

```
from instruction_graph.interactive.InteractiveManager import
↪  InteractiveManager, BuilderPhrases
from instruction_graph.example.DefaultMemory import DefaultMemory
from instruction_graph.example.ExamplePrimitiveLibrary import
↪  ExamplePrimitiveLibrary
from instruction_graph.interactive.utils import regex

memory_obj = DefaultMemory()
library = ExamplePrimitiveLibrary()

builder_phrases = BuilderPhrases()
builder_phrases.confirm_pos = regex("(Yes|Yeah|Righto)")
builder_phrases.confirm_neg = regex("Nope")
builder_phrases.agent_start = "Get ready! It's interactive time."

im = InteractiveManager(library=library, memory=memory_obj,
↪  phrases=builder_phrases)
```

Attribute	Default	Description
teach_you	`regex("i will teach you to (.*)")`	human instruction to IM to begin learning a graph with name in the first captured group
confirm_pos	`regex("y(?:es|eah)?")`	human affirmative
confirm_neg	`regex("no")`	human negative
teaching_done	`regex("done (?:learning|teaching)")`	human instruction to IM to save and finish graph currently under construction
h_if_cond	`regex("if (?P<neg>(?:not|don’?t) )?(?P<cmd>.*)")`	add a conditional primitive specified by the `<cmd>` capturing group as an IF condition to the graph. If the `<neg>` capturing group is present, negate it.
h_while_cond	`regex("(?:while|loop) (?P<neg>(?:not|don’?t) )?(?P<cmd>.*)")`	add a conditional primitive specified by the `<cmd>` capturing group as a WHILE/LOOP condition to the graph. If the `<neg>` capturing group is present, negate it.
h_else	`regex("else")`	add an ELSE node
h_end_if	`regex("end if")`	add an END IF node
h_end_loop	`regex("end loop")`	add an END LOOP node
run_ig	`regex("run (.*)")`	if in state WAITING, execute the saved graph specified by the capturing group. if in the process of building a graph, add a *run_ig* node to run a graph with this name
agent_start	`"Beginning interactive graph runner and builder."`	IM outputs this at startup
exec_prim_success	`""`	when a primitive is successfully *executed*, the IM responds with this
exec_prim_fail	`"Could not find or run primtive <prim_id>"`	What to output when the IM believes the human specified a primitive but it can’t find the primitive specified
exec_graph_success	`""`	the IM outputs this after successfully executing a graph
exec_graph_fail	`"Could not find or run graph <prim_id>"`	What to output when the IM believes the human specified a graph but it can’t find the graphspecified in the directory
build_new_graph	`"I will learn to <name>?"`	when the IM believes that the human has asked it to learn a new graph, it outputs a request for confirmation
yes_learn	`"Ok, I am ready to learn.  What is first?"`	The IM outputs this at the beginning of learning a graph.
no_learn	`"Ok"`	The IM outputs this after receiving a negative response to checking whether it should learn a graph
unclear_confirm	`"I don’t understand.  Please name a primitive to add or tell me I’m done."`	The IM outputs this when it cannot process the human response after the IM asks for a yes/no response.
unclear_learn	`"I don’t understand.  Please name a primitive to add or tell me I’m done."`	The IM outputs this if it doesn’t understand text input during graph construction.
confirm_new_primitive	`"I should <cond> <name>"`	During graph construction, the IM asks for confirmation for adding a primitive using this construction. `<name>` will be replaced with the name of the primitive (including parameters), and `<cond>` will be replaced by any of the following if required: IF, WHILE, NOT
new_primitive_confirmed_pos	`"Ok, what’s next?"`	When the human responds confirming adding a node, the IM outputs this.
new_primitive_confirmed_neg	`"Ok, what’s next?"`	When the human responds rejecting confirmation of adding a node, the IM outputs this.
confirm_add_run_graph_name	`"run the graph <name>"`	The IM uses this construction to confirm adding a *run_ig* node
new_run_graph_not_exist	`"The graph you are requesting I add does not seem to exist.  I checked for it at <location>."`	when attempting to add a *run_ig* node to a graph, this is output if the graph is not fond
done_building_graph	`"I have learned <name>."`	When the IM saves a graph and exits graph construction, it outputs this.

Table B.1: Builder Phrases

B.6 Credits

The Instruction Graph library has been created by

- Aaron M. Roth
- Çetin Meriçli
- Steven D. Klee

Appendix C

Documentation for Vehicle Intersection Open Source AI Gym Environment

Vehicle Intersection (VI) is an AI Gym[1] compatible reinforcement learning environment. It is a “Toy Text” environment. It is highly configurable and difficult.

[1]<https://gym.openai.com/>

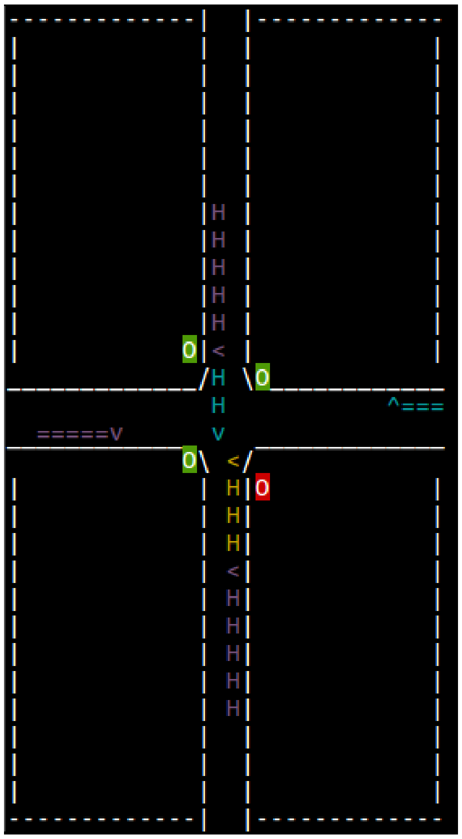


Figure C.1: VI rendering example

C.1 Description

Vehicle Intersection simulates a four-way vehicle intersection with stoplights. Vehicles spawn at the edges of the map and approach the intersection at the center, driving on two-land roads. Each approach to the intersection has a stoplight that can be red or green. The agent controls these four traffic lights, and action involve turning the lights red or green.

Vehicles move at a fixed speed when unimpeded. If they are blocked by a vehicle in front of them going in the same direction, they stop. If they are blocked by a vehicle in front of them going in a different direction, there is a collision. If a vehicle approaches the edge of the intersection and encounters a green light, it continues moving, if it approaches the edges of an intersection and encounters a red light, it stops and waits until the light is green, at which point it move forward.

Additionally, each vehicle has an attribute called a "turn signal" indicating whether it will turn, go left, or go right at the intersection.

Each episode runs either until a configurable number of vehicles spawn and exit the intersection safely or until a collision occurs.

The agent receives a penalty of -1 when a vehicle is forced to wait at a red light (per vehicle, per timestep). A vehicle stuck behind another vehicle waiting at a red light also incurs this penalty. The agent receives a penalty of -1000 when a collision occurs. An optimal episode has at most a reward of 0 (which may not always be possible depending how vehicles spawn). The goal becomes to reduce waiting time for vehicles while also avoiding collisions.

C.2 Action Space

There are five "Action Sets" which correspond to action spaces. You can import them as follows:

```
from vi_env.VehicleIntersectionActionSets import SetByRoad,
↪  SetByLight, ToggleRoad, ToggleLight, SetExplicitly
```

When instantiated, each action set has a `describe()` method that gives a description (see **Example** section). Actions are passed in to the step function as integers.

The five action spaces available by default are described here as well, listed in order of increasing action space size:

C.2.1 ToggleRoad()

1. wait
2. toggle lights on vertical road
3. toggle lights on horizontal road

C.2.2 SetByRoad()

1. wait
2. make lights on vertical road green
3. make lights on vertical road red
4. make lights on horizontal road green
5. make lights on horizontal road red

C.2.3 ToggleLight()

1. wait
2. toggle lights on left approach
3. toggle lights on upper approach
4. toggle lights on right approach

5. toggle lights on lower approach

C.2.4 SetByLight()

1. wait
2. makes light on left approach green
3. makes light on left approach red
4. makes light on upper approach green
5. makes light on upper approach red
6. makes light on right approach green
7. makes light on right approach red
8. makes light on down approach green
9. makes light on down approach red

C.2.5 SetExplicitly()

1. wait
2. set lights to Red, Red, Red, Red (Left, Up, Right, Down)
3. set lights to Red, Red, Red, Green
4. set lights to Red, Red, Green, Red
5. set lights to Red, Red, Green, Green
6. set lights to Red, Green, Red, Red
7. set lights to Red, Green, Red, Green
8. set lights to Red, Green, Green, Red
9. set lights to Red, Green, Green, Green
10. set lights to Green, Red, Red, Red
11. set lights to Green, Red, Red, Green
12. set lights to Green, Red, Green, Red
13. set lights to Green, Red, Green, Green
14. set lights to Green, Green, Red, Red
15. set lights to Green, Green, Red, Green
16. set lights to Green, Green, Green, Red
17. set lights to Green, Green, Green, Green :

C.3 Observation Space

The observation space is represented by a vector of length `4+4*V`, where `V` is the maximum the number of vehicles that are allowed to be on the map at a single time.

The first four elements indicate which lights are red or green (0 for red, 1 for green) in the following order: Left, Up, Right, Down.

Each set of the next four elements indicates a vehicle on the map. The four elements in order are as follows:

1. Approach Road: 0-3 for Left, Upper, Right, Lower
2. Position of Front of Vehicle: for an intersection of size 2 and road length of `R` this can range from `0` to `R+2`. 0 means it is about to exit the intersection, 2 would mean it is about to enter the intersection, and a higher number is some distance farther away from the intersection.
3. Length of vehicle: the remaining length of a the vehicle. Note that when a vehicle partially exits the intersection, it will be represented as a smaller vehicle about to exit.
4. Turn Signal: -1 for left, 0 for straight, and 1 for right

C.4 Requirements

VI requires Python 3 and the following Python libraries, all of which available on pip:

- gym 0.10.5+
- six 1.11.0+
- numpy 1.15.4+
- bitarray 0.8.3+

C.5 Setup

In the **vi_env** folder find the three files with the code for Vehicle Intersection. You can take the **vi_env** folder and drop it into your project and import it as shown in the **Usage** or **Example** section, or just copy in those three files. Please maintain attribution information.

C.6 Usage

See an example usage below:

```
from vi_env.VehicleIntersection import VehicleIntersection

env = VehicleIntersection()
env.reset()
action = env.action_space.sample()
env.step(action)
env.render()
```

If you like, you can also specify certain settings in the environment.

Some useful ones may be the action set, the maximum vehicles on the map, and the road length.

```python
from vi_env.VehicleIntersection import VehicleIntersection
from vi_env.VehicleIntersectionActionSets import SetByLight

env = VehicleIntersection(action_set=SetByLight(),
                          max_vehicles_on_map=4,
                          road_length=20)
env.reset()
action = env.action_space.sample()
env.step(action)
env.render()
```

A Vehicle Intersection environment has the standard methods of `reset()` and `step(action)`, as well as a `render()` method which can be run in a terminal to output a visual representation of the state.

C.7 Example

Included in this repository is the `disp_env.py` script, which runs the environment with random actions. Run it with `python3 disp_env.py` in the terminal to see a representation of the environment over a sequence of timesteps with random actions.

C.8 Settings

`VehicleIntersection()` can be instantiated with a number of parameters. The full list of customizable parameters is explained here.

- action_set - which of the action sets from to use (ToggleRoad(), ToggleLight(), SetByRoad(), SetByLight(), SetExplicitly()), default: ToggleRoad()
- road_length - length of the road before the intersection. (intersection length is 2.) default: 14
- use_turn_signals - boolean. if False, cars only go straight. default: True
- turn_signal_dist - 3-tuple. If using turn signals, probabilities of each occurring in a given vehicle. (Left, None [Straight], Right) default: (0.33, 0.34, 0.33)
- max_vehicles_on_map - when this number of vehicles are on the map, do not spawn additional vehicles until at least one exits the intersection. **Note: This parameter affects the size of the observation space.** default: 8

- vehicle_types - an OrderedDict of (string → 2-Tuple(int, int)) pairs. This corresponds to ("vehicle name" → ("length min", "length max")). default: `OrderedDict([("car", (4, 4)), ("van", (6, 6)), ("truck", (8, 12))])`
- vehicle_dist - Tuple of length equal to size of **vehicle_types**. Probability of each vehicle type spawning each time a vehicle is to spawn. default: (0.4, 0.35, 0.25)
- vehicle_colors - OrderedDict of (string → string) pairs indicating what color each vehicle type should be in the rendering. default: OrderedDict([("car", "yellow"), ("van", "magenta"), ("truck", "cyan")])
- vehicle_speed - the distance an unimpeded vehicle will move each timestep. default: 1
- spawn_interval - 2-Tuple indicating minimum and maximum time to wait between spawning vehicles. default: (2, 7)
- vehicles_to_spawn_per_episode - the number of vehicles to spawn per episode. affects episode length. default: 10
- lights_init - starting configuration of the lights. The default is all green. default: (1, 1, 1, 1)
- waiting_penalty - default: -1
- collision_penalty - default: -1000

Appendix D

Gridsearch Bounds Used and Hyperparameters Found When Learning Decision Tree Policies

In Section 7.5, results were shown for various methods and environments. In each case, a gridsearch was performed to find the optimal hyperparameters. The bounds of the gridsearch for pCQI, CQI, and CQI2 are shown in Table D.1. The bounds of the gridsearch for Pyeatt Method are shown in Table D.2.

Environment	**Method**	**Bounds**		
		α	H_S	*numSplits*
Robot Navigation A	pCQI	0.0001 to 0.3	1 to 10,000,000	n/a
	CQI	0.0001 to 0.3	1 to 1,000,000,000	2 to 9
	CQI2	0.001 to 0.3	0.1 to 10,000,000,000	2 to 10
Robot Navigation B	pCQI	0.0001 to 0.2	0.01 to 1,000	n/a
	CQI	0.0001 to 0.2	0.01 to 1,000	2 to 7
	CQI2	0.001 to 0.4	0.05 to 100,000	2 to 9
Vehicle Intersection	pCQI	0.01 to 0.4	0.00005 to 1000	n/a
	CQI	0.01 to 0.4	0.00001 to 1000	2 to 7
	CQI2	0.01 to 0.5	0.00001 to 10,000	2 to 9

Table D.1: Gridsearch bounds for pCQI, CQI, and CQI2

Environment	**Method**	**Bounds**	
		α	History List Min.
Robot Navigation A	Pyeatt Method	0.01 to 1	1,000 to 25,000
Robot Navigation B		0.1 to 0.8	5,000 to 30,000
Vehicle Intersection		0.01 to 0.8	1000 to 15,000

Table D.2: Gridsearch bounds for Pyeatt Method

Env:	**Robot Navigation A**				**Robot Navigation B**			
Method	α	H_S	$numSplits$	History List Min.	α	H_S	$numSplits$	History List Min.
PM	0.9	-	-	3,000	0.2	-	-	8,000
pCQI (R)	0.001	10,000	2	-	0.001	10	2	-
pCQI (S)	0.001	100,000	2	-	0.001	10	2	-
CQI (R)	0.005	1,000	4	-	0.001	10	6	-
CQI (S)	0.001	100,000	5	-	0.001	10	4	-
CQI2 (R)	0.1	100,000	9	-	0.1	1	7	-
CQI2 (S)	0.2	1,000,000,000	8	-	0.01	0.1	8	-

Env:	**Vehicle Intersection**			
	Tree Size		**Avg. Reward/Ep.**	
Method	α	H_S	$numSplits$	History List Min.
PM	0.6	-	-	10,000
pCQI (R)	0.3	0.001	2	-
pCQI (S)	0.1	0.001	2	-
CQI (R)	0.3	0.0001	6	-
CQI (S)	0.2	0.01	4	-
CQI (S)	0.3	0.001	5	-
CQI2 (R)	0.3	0.0001	2	-
CQI2 (S)	0.01	0.01	7	-
CQI2 (S)	0.1	0.01	9	-

Table D.3: Best hyperparameters found for various RL decision tree policy methods across environments

In Section 7.5, results were shown for various methods and environments. In each case, these results are using the optimal hyperparameters found in the grid search. The optimal hyperparameters for each entry in Table 7.1 is shown in Table D.3 above.

Appendix E

TAIGs for Additional Subtask Graphs for GPSR

In this section please find additional examples of GPSR sub-task TAIGs. All primitives used in these examples can be found in Appendix E.

Listing E.1: TAIG for "Discover the Name of a Person in a Location and Report it" ("nameperson") GPSR Subtask.

```
START GRAPH
queue_subtask_statement ``planner_config/planner/subtask''
say_with_args   ``I am going to learn who is in %s''
say_with_args   ``I am going to go to %s''
go_to_location  None
say_with_args   ``I am in the %s''
say     ``Let me look around for who is here.''
find_any_person [None, None]
IF person_found
        begin_frame_listen      ``names''
        say     ``Hello human, what is your name?''
        WHILE NOT frame_heard
        END LOOP
        learn_name_from_frame
        say_with_args ``Your name is %s.''
        say_with_args ``Hello %s, it is nice to met you.  I will
           ↪ go tell the operator that you are here.''
        go_to_location ``designated_start''
        say_with_args ``I went to the room you asked me to go to
           ↪ and I found %s.''
ELSE
        say     ``I cannot find anyone here.  I will go back.''
        go_to_location ``designated_start''
        say_with_args ``I did not find anyone at %s.''
END IF
say     ``Task Complete''
END GRAPH
```

Listing E.2: TAIG for “Go to a Location and say Something to a Person” (“talk-nameplace”) GPSR Subtask.

```
START GRAPH
queue_subtask_statement ``planner_config/planner/subtask''
say_with_args   ``I am going to find %s at the %s and tell %s.''
go_to_location  None
say_with_args   ``I am looking for %s.  I am at the %s.  I will
   ↪ tell %s once I find the person.''
find_any_person [None, None]
IF person_found
        IF is_answer_question_gpsr
                say     ``Hello!  Nice to meet you.  My operator
                   ↪ told me to anwer a question from you.
                   ↪ Please ask your question.''
                begin_frame_listen      ``predefined''
                WHILE NOT frame_heard
                END LOOP
                sa_question_gpsr
                say_with_args ``The question Iam to answer is %s
                   ↪ .''
                answer_question_gpsr
                say_with_args ``The answer to your question is %s
                   ↪ ''
                say     ``Thank you.  I will return.''
        ELSE
                set_answer_gpsr
                say_with_args   ``Hello!  Nice to meet you.  My
                   ↪ operator asked me to tell you %s.''
                say     ``I will go back now.''
        END IF
        go_to_location  ``designated_start''
        say     ``I did not find anyone.  The task is done.''
END IF
END GRAPH
```

Listing E.3: TAIG for “Find a Person of a Certain Attribute at a Location and say Something” (“talkggpplace”) GPSR Subtask.

```
START GRAPH
queue_subtask_statement ``planner_config/planner/subtask''
say_with_args   ``I am going to find a person who is %s at the %s
    ↪  and tell %s.''
go_to_location   [None]
say_with_args   ``I am looking for a person who is %s.  I am at
    ↪ the %s.  I will tell %s once I find the person.''
find_any_person [None, None]
IF person_of_interest_found
        IF is_answer_question_gpsr
                say     ``Hello! Nice to meet you.  My operator
                    ↪ told me to answer a question from you.
                    ↪ Please ask your question.''
                begin_frame_listen      ``predfined''
                WHILE NOT frame_heard
                END LOOP
                sa_question_gpsr
                say_with_args ``The answer to your question is %s
                    ↪ .''
                say     ``Thank you. I will return''
        ELSE
                set_answer_gpsr
                say_with_args   ``Hello!  Nice to meet you.  My
                    ↪ operator asked me to tell you %s.''
                say     ``I will go back now.''
        END IF
        go_to_location ``The task is done.''
ELSE
        say_with_args ``I cannot find anyone who is %s here at %s
            ↪  to tell %s.  I will go back.''
        go_to_location  ``designated_start''
        say_with_args   ``I did not find anyone who is %s at %s
            ↪ to tell %s. The task is done.''
END IF
END GRAPH
```

Listing E.4: TAIG for “Bring an object from a location.” (“retrieveobjfrom”) GPSR Subtask.

```
START GRAPH
queue_subtask_statement ``planner_config/planner/subtask''
say_with_args   ``I am going to retrieve %s from %s.''
go_to_location  [None]
say_with_args   ``I am looking for %.  I am in %s now.''
search_for_catobj       None
IF catobj_found
        say_with_args   ``I found %s in %s.''
        approach_obj
        point
        say     ``It doesn't look like something that I can pick
           ↪ up.  I am going to look for someone to help me.''
        put_down_arm
        init_head
        find_person_in_place
        IF person_found
                set_helper_info
                say_with_args   ``Excuse me person wearing a %s
                   ↪ shirt, or anyone in this room.  Would you
                   ↪ helpl me pick up the %s here in teh %s?  I
                   ↪ can show you wehre to bring it.  If you do
                   ↪ this for me, my team's humans will give you
                   ↪  candy later.  Please answer yes or no.''
                begin_frame_listen      ``yesno''
                WHILE NOT helper_response_heard
                END LOOP
                IF positive_response
                        say     ``Ok. Please pick up the object
                           ↪ and follow me.''
                        go_to_location  ``designated_start''
                        say_with_args   ``With the help of a
                           ↪ person wearing a %s shirt who
                           ↪ should have followed me here, I
                           ↪ brought the %s in the %s.  The task
                           ↪  is done.''
                ELSE
                        ``Ok.  Thanks for letting me know.''
                        go_to_location  ``designated_start''
                        say_with_args   ``The person wearing a %s
                           ↪  shirt would not help me bring the
                           ↪ %s in the %s over, although I did
```

```
                                ↪ find it.  Sorry.''
                    END IF
          ELSE
                    say ``It seems that no one is in here.  I will go
                        ↪  back.''
                    go_to_location ``designated_start''
                    say     ``I found the object, but I couldn't find
                        ↪  anyone to help me bring it over.''
          END IF
ELSE
          say     ``I cannot find the object here.  I will go back
              ↪ .''
          go_to_location  ``designated_start''
END IF
END GRAPH
```

Listing E.5: TAIG for "Find an object in a room." ("findobjcat") GPSR Subtask.

```
START GRAPH
queue_subtask_statement ``planner_config/planner/subtask''
say_with_args   ``I am going to find the %s int he %s.''
go_to_location  [None]
say_with_args   ``I am looking for the %s.  I am in the %s now.''
search_for_catobj
IF catobj_found
          IF is_cat
                    say_with_args   ``I see %s which belongs to the
                        ↪ category %s here in the %s.  I will go back
                        ↪  and report to my operator.''
                    put_down_arm
                    init_head
                    go_to_location  ``designated_start''
                    say_with_args   ``I found %s which belongs to
                        ↪ theh category %s in the %s.  The task is
                        ↪ done.''
          ELSE
                    say_with_args   ``I see the %s here in the %s.  I
                        ↪  will go back and report to my operator.''
                    put_down_arm
                    init_head
                    go_to_location  ``designated_start''
                    say_with_args   ``I found the %s in the %s.  The
                        ↪ task is done.''
          END IF
```

```
ELSE
        say     ``I cannot find the object here.  I will go back
           ↪ .''
        init_head
        go_to_location  ``designated_start''
        say_with_args   ``I did not find the %s in the %s.  The
           ↪ task is done.''
END GRAPH
```

Listing E.6: TAIG for “Report Which Object or Object-in-a-Category has a Certain Property at a Location” (“findobjcatextreme”) GPSR Subtask.

```
START GRAPH
queue_subtask_statement ``planner_config/planner/subtask''
say_with_args   ``I am going to find the %s %s in the %s.''
go_to_location  [None]
say_with_args   ``I am looking for the %s %s here.  I am in the %
    ↪ s now.''
search_extreme_objcat   [None, None]
IF found_objcat_extreme
        say_with_args   ``I have found the %s %s here in the %s.
            ↪ I will go back and tell the operator.''
        go_to_location  ``designated_start''
        say_with_args   ``I have found the %s %s in the %s.  The
            ↪ task is done.''
ELSE
        say     ``I cannot find any of the designated object here
            ↪ .  I will go back and tell the operator.''
        go_to_location  ``designated_start''
        say_with_args   ``I could not find the %s %s in the %s
            ↪ because none of them was there.  The task is done
            ↪ .''
END IF
END GRAPH
```

Listing E.7: TAIG for “Find Three Objects or Categories of Objects of a Certain Nature at a Certain Location” (“findthreecatextreme”) GPSR Subtask.

```
START GRAPH
queue_subtask_statement ``planner_config/planner/subtask''
say_with_args   ``I am going to find three %s %s at the %s.''
go_to_location  [None]
say_with_args   ``I am looking for three %s %s here.  I am at the
    ↪ %s now.''
search_extreme_objcat   [None, None]
IF found_objcat_extreme
        say     ``I have found the objects here.  I will go back
            ↪ and tell the operator.''
        go_to_location  ``designated_start''
        say_with_args   ``I have found the %s %s in the %s.  The
            ↪ task is done.''
ELSE
        say     ``I cannot find any of the designated object here
            ↪ .  I will go back and tell the operator.''
        go_to_location  ``designated_start''
        say_with_args   ``I could not find the %s %s in the %s
            ↪ because none of them was there.  The task is done
            ↪ .''
END IF
END GRAPH
```

Listing E.8: TAIG for “Find a Person of a Certain Attribute in a Room” (“find$_person_attr''$)$GPSRSubtask$.

```
START GRAPH
queue_subtask_statement ``planner_config/planner/subtask''
say_with_args   ``I am going to find a person %s in the %s.''
go_to_location  [None]
say_with_args   ``I am looking for a person %s. I am in the %s
   ↪ now.''
find_any_person [None, None]
IF person_of_interest_found
        say_with_args   ``I found a person %s in the %s.''
        go_to_location  ``designated_start''
        say_with_args   ``I found a person %s in the %s. The task
            ↪  is done.''
ELSE
        say_with_args   ``I cannot find any person %s in the %s
            ↪ here.  I will go back.''
        go_to_location  ``designated_start''
        say_with_args   ``I did not find any person %s in the %s.
            ↪  The task is done.''
END IF
END GRAPH
```

Listing E.9: TAIG for “Report how many people in a room satisfy certain criteria.” (“countgp”) GPSR Subtask.

```
START GRAPH
queue_subtask_statement ``planner_config/planner/subtask''
say_with_args   ``I am going to count how many people who are %s
   ↪ in the %s.''
go_to_location  [None]
say_with_args   ``I am going to look for all the people who are %
   ↪ s here.  I am in the %s now.''
count_ppl_with_attr     [None]
say_with_args   ``I have found %s who are %s in the %s.  I will
   ↪ go back and report the number.''
```

```
go_to_location   ``designated_start''
say_with_args    ``I have found %s who are %s in the %s.''
END GRAPH
```

Listing E.10: TAIG for “Take an Object From You, and Deliver It to a Location” (“placeobj”) GPSR Subtask.

```
START GRAPH
queue_subtask_statement ``planner_config/planner/subtask''
say_with_args   ``I am going to take the %s from you and deliver
   ↪ it to the %s.''
say     ``It does not look like something that I can pick up.
   ↪ Would you be able to hold on to it? I can guide you to the
   ↪ destination.  Please answer yes or no.''
begin_frame_listen      ``yesno''
WHILE NOT helper_response_heard
END LOOP
IF positive_response
        say_with_args   ``Great. Please take the %s and follow me
           ↪  to the %s.''
        go_to_location  [None]
        say_with_args   ``Thank you for brining the %s over. We
           ↪ are at the %s now.  The task is done.''
ELSE
        say_with_args   ``Alright. I will find someone else to
           ↪ ehlp me take the %s to the %s.''
        find_person_in_place
        IF person_found
                set_helper_info
                say_with_args   ``Excuse me person wearing a %s
                   ↪ shirt, or anyone in this room.  Would you
                   ↪ help me take the %s from the operator to
                   ↪ the %s?''
                begin_frame_listen      ``yesno''
                WHILE NOT helper_response_heard
                END LOOP
                IF positive_response
                say     ``Ok. Please take the object from the
                   ↪ operator and follow me.''
                go_to_location  [None]
                say     ``We have reached our destination. Thank
                   ↪ ou for helping. I will go back.''
                go_to_location  ``designated_start''
        ELSE
                say     ``Ok. Thanks for letting me know.''
                go_to_location  ``designated_start''
                say     ``I could not find anyone to help me take
                   ↪  the %s to the %s.  Sorry.''
        END IF
END IF
END GRAPH
```

Listing E.11: TAIG for “Find an Object in a Location, and Take it to Another Location” (“takeplaceobj”) GPSR Subtask.

```
START GRAPH
queue_subtask_statement ``planner_config/planner/subtask''
say_with_args   ``I am going to take the %s from %s to %s.''
go_to_location  [None]
say_with_args   ``I am looking for %s. I am in the %s now.  I
   ↪ will go to the %s later.''
search_for_catobj       [None]
IF      catobj_found
        say_with_args   ``I found %s in the %s. I will take it to
           ↪  the %s.''
        say     ``It doesn't look like something that I can pick
           ↪ up.  I am going to look for someone to help me.''
        put_down_arm
        init_head
        find_person_in_place
        IF person_found
                set_helper_info
                say_with_args   ``Excuse me person wearing a %s
                   ↪ shirt, or anyone in this room.  Would you
                   ↪ help me take the %s here in the %s to the %
                   ↪ s?  Please answer yes or no.''
                begin_frame_listen      ``yesno''
                WHILE NOT helper_response_heard
                END LOOP
                IF positive_response
                        say     ``Ok. Please pick up the object
                           ↪ and follow me.''
                        go_to_location  [None]
                        say     ``We have reached our destination
                           ↪ .  Please leave the object here.
                           ↪ Thank you.''
                ELSE
                        say     ``Ok. Thanks for letting me know.
                           ↪   I will go back.''
                        go_to_location ``designated_start''
                        say_with_args   ``The person wearing a %s
                           ↪  shirt would not help me take the %
```

```
                                ↪ s from the %s to the %s, although I
                                ↪ did find it. Sorry.''
                END IF
        ELSE
                say        ``It seems tha tno one is in here. I
                   ↪ will go back.''
                go_to_location  ``designated_start''
                say_with_args   ``I found the %s in the %s, but I
                   ↪ could not find anyone to help me take it
                   ↪ to the %s. Sorry.''
        END IF
ELSE
        say        ``I cannot find the object here. I will go back
           ↪ ''
        go_to_location  ``designated_start''
END IF
END GRAPH
```

Listing E.12: TAIG for “Find a person at a location, and report back regarding one of their attributes.” (“ngpperson”) GPSR Subtask.

```
START GRAPH
queue_subtask_statement ``planner_config/planner/subtask''
say_with_args   ``I am going to a report a person's %s at the %s
   ↪ ''
go_to_location  [None]
say_with_args   ``I am going to look for a person and report
   ↪ their %s. I am at the %s now.''
find_person_in_place
IF is_report_name
        IF person_found
                begin_frame_listen ``name''
                say     ``Hello human, what is your name?''
                WHILE NOT frame_heard
                END LOOP
                learn_name_from_frame
                say_with_args   ``Your name is %s.''
                say_with_args   ``Hello %s, it is nice to meet
                   ↪ you.  I will go tell the operator that you
                   ↪ are here.''
                go_to_location ``designated_start''
                say_with_args   ``I went to the room you asked me
                   ↪  to go to and I found %s.''
        ELSE
                say     ``I cannot find anyone here.  I will go
                   ↪ back.''
                go_to_location  ``designated_start''
                say_with_args   ``I went to the room you asked me
                   ↪  to go to but I could not find anyone there
                   ↪ .''
        END IF
ELSE
        report_human_attr       [None]
        IF human_attr_found
                say_with_args   ``I am at the %s.  I have found a
                   ↪  person who is %s here.  I will go back and
                   ↪  report to my oeprator.''
                go_to_location  ``designated_start''
                say_with_args   ``The person I found at the %s
                   ↪ was %s.  The task is done.''
        ELSE
                say     ``I cannot find anyone here.''
```

```
                    go_to_location  ``designated_start''
                    say     ``I did not find anyone at the designated
                        ↪ location.  The task is done.''
          END IF
END IF
END GRAPH
```

Listing E.13: TAIG for "Find three objects of a certain category in a room." ("findthreecat") GPSR Subtask.

```
START GRAPH
queue_subtask_statement ``planner_config/planner/subtask''
say_with_args ``I am going to find three objects of the category
    ↪ %s in the %s.''
go_to_location   [None]
say_with_args   ``I am going to find the objects of the category
    ↪ %s. I am in the %s now.''
count_objcats    [None]
say_with_args   ``I have found %s %s here at the %s. I will go
    ↪ back and report to the operator.''
go_to_location   ``designated_start''
say_with_args   ``I found %s %s at the %s in total.  The task is
    ↪ done.''
END GRAPH
```

Listing E.14: TAIG for "Count how many of an object or category there are." ("count$_o bjcat'')GPSRSubtask$.

```
START GRAPH
queue_subtask_statement ``planner_config/planner/subtask''
say_with_args   ``I am going to count how many %s there are at %s
    ↪ .''
go_to_location   [None]
say_with_args   ``I am looking for all the %s here.  I am at the
    ↪ %s now.''
count_objcats    [None]
say_with_args   ``I have found %s %s here at the %s.  I will go
    ↪ back and report to the operator.''
init_head
go_to_location   ``designated_start''
say_with_args   ``I found %s %s at the %s in total.  The task is
    ↪ done.''
END GRAPH
```

Listing E.15: TAIG for “Get the Object at the Default Location (Not Given Location to Search For)” (“retrieveobj”) GPSR Subtask.

```
START GRAPH
queue_subtask_statement ``planner_config/planner/subtask''
say_with_args   ``I am going to find the %s.''
get_obj_loc     [None]
say_with_args   ``The default location of the %s is %s.  I am
   ↪ going to look for it there.''
go_to_location  [None]
say_with_args   ``I am looking for %s. I am at the %s now.''
search_for_catobj       [None]
IF catobj_found
        say_with_args   ``I found %s at the %s.''
        approach_obj    [None]
        point
        say     ``It doesn't look like something that I can pick
           ↪ up.  I am going to look for someone to help me.''
        put_down_arm
        init_head
        find_person_in_place
        IF person_found
                set_helper_info
                say_with_args   ``Excuse me person wearing a %s
                   ↪ shirt, or anyone in this room.  Would you
                   ↪ help me pick up the %s here at the %s? I
                   ↪ can show you where to bring it.  If you do
                   ↪ this for me, my team's humans will give you
                   ↪  candy later.  Please answer yes or no.''
                begin_frame_listen      ``yesno''
                WHILE NOT helper_response_heard
                END LOOP
                IF positive_response
                        say     ``Ok. Please pick up the object
                           ↪ and follow me.''
                        go_to_location  ``designated_start''
                        say_with_args   ``With the help of a
                           ↪ person wearing a %s shirt who
                           ↪ should have followed me here, I
                           ↪ brought the %s at the %s.  The task
                           ↪  is done.''
                ELSE
                        say     ``Ok. Thanks for letting me know
                           ↪ .''
```

```
                    go_to_location  ``designated_start''
                    say_with_args   ``The person wearing a %s
                       ↪  shirt would not help me bring the
                       ↪ %s at the %s over, although I did
                       ↪ find it.  Sorry.''
                END IF
        ELSE
                say     ``It seems that no one is in here.  I
                   ↪ will go back.''
                go_to_location  ``designated_start''
                say     ``I found the object, but I couldn't find
                   ↪  anyone to help me bring it over.''
        END IF
ELSE
        say     ``I cannot find the object here.  I will go back
           ↪ .''
        go_to_location  ``designated_start''
        say_with_args   ``I could not find the %s at the %s.
           ↪ Sorry.''
END IF
END GRAPH
```

Listing E.16: TAIG for “Take the Object and Deliver It to a Person at a Beacon” (“deliverobjname”) GPSR Subtask.

```
START GRAPH
queue_subtask_statement ``planner_config/planner/subtask''
say_with_args   ``I am going to find %s and deliver the %s at the
   ↪  %s.''
say     ``It does not look like something that I can pick up.
   ↪ Would you help me deliver it? If you do this for me, my
   ↪ team's humans will give you candy later.  Please answer yes
   ↪  or no.''
begin_frame_listen      ``yesno''
WHILE NOT helper_response_heard
END LOOP
IF positive_response
        say     ``Ok. Please take the object and follow me.''
        go_to_location  [None]
        say_with_args   ``I am looking for %s to deliver the %s.
           ↪  I am at the %s now.''
        find_any_person
        IF person_found
                say_with_args   ``Hello!  You must be %s.  My
```

```
                    ↪ operator helped me bring the %s here at the
                    ↪  %s. Please take it from my operator. ``
                say     ``the task is done.  Thank you for
                    ↪ helping me bring it over.  I will go back
                    ↪ .''
                go_to_location  ``designated_start''
        ELSE
                say_with_args   ``I cannot find %s here to
                    ↪ deliver the %s to. I am at the %s.  I will
                    ↪ go back.''
                go_to_location  ``designated_start''
        END IF
ELSE
        say_with_args   ``Sorry.  I am not able to find % and
            ↪ deliver the %s and the %s, because I cannot pick it
            ↪  up.''
END IF
END GRAPH
```

Listing E.17: TAIG for "Meet a Person at a Beacon, and Guide Them To Another Beacon" ("guide") GPSR Subtask.

```
START GRAPH
queue_subtask_statement ``planner_config/planner/subtask''
say_with_args   ``I am going to guide %s from %s to %s.''
go_to_location  [None]
say_with_args   ``I am looking for %s. I am at %s. I will go to %
    ↪ s later.''
say     ``Let me look around for who is here.''
find_any_person [None, None]
IF person_found
        say_with_args   ``Hello.  You must be %s.  My operator
            ↪ told me to guide you from %s to %s.''
        say     ``Would you follow me? I can lead you to your
            ↪ destination.  Please answer yes or no.''
        begin_frame_listen      ``yesno''
        WHILE NOT helper_response_heard
        END LOOP
        IF positive_response
                say_with_args   ``OK %s.  We are at the %s now.
                    ↪ We are heading to the %s.''
                go_to_location  [None]
                say     ``We have reached our destination.  I
                    ↪ will go back to my operator.''
                go_to_location  ``designated_start''
                say_with_args   ``I have guided %s from %s to %s.
                   ↪   The task is done.''
        ELSE
                say     ``Alright.  Thanks for letting me know.
                   ↪ I will go back.''
                go_to_location  ``designated_start''
                say_with_args   ``I have found %s at the %s who
                   ↪ refused to follow me to %s.  The task is
                   ↪ done.''
        END IF
ELSE
        say_with_args   ``I cannto find %s here to guide from %s
           ↪ to %s.  I wil lreturn.''
        go_to_location  ``designated_start''
        say     ``I did not find the person you told me to look
           ↪ for. The task is done.''
END IF
END GRAPH
```

Listing E.18: TAIG for “Meet a Person and Guide Them (When Locations are not Initially Specified)” (“guidenothing”) GPSR Subtask.

```
START GRAPH
queue_subtask_statement ``planner_config/planner/subtask''
say_with_args   ``I am going to guide %s.''
say     ``I don't know where the person is and where to guide
   ↪ them. Would you be able to specify where the person is and
   ↪ where to guide them?  Please first say the location of the
   ↪ person.  For example, bedroom.''
begin_frame_listen      ``location''
WHILE NOT frame_heard
END LOOP
add_location    [0]
say_with_args   ``Alright.  I am going to guide %s at the %s.''
say     ``Now please say the location of the destination.  For
   ↪ example, kitchen.''
begin_frame_listen      ``location''
WHILE NOT frame_heard
END LOOP
add_location    [1]
say_with_args   ``Thank you! I am going to guide %s at the %s to
   ↪ the %s.''
go_to_location  [None]
say_with_args   ``I am looking for %s now.  I am at the %s.  I
   ↪ will go to %s later.''
find_person_in_place
IF person_found
        say_with_args   ``Hello %s. My operator told me to guide
           ↪ you from %s to %s.''
        say     ``Would you follow me? Please answer yes or no.''
        begin_frame_listen      ``yesno''
        WHILE NOT frame_heard
        END LOOP
        IF positive_response
                say     ``Ok.  Please follow me.''
                go_to_location  [None]
                say     ``We have reached our destination.  Thank
                   ↪  you!  I will go back now.''
                go_to_location  ``designated_start''
        ELSE
                say     ``Alright.  Thanks for letting me know.
                   ↪ I will go back now.''
                go_to_location  ``designated_start''
        END IF
ELSE
        say     ``I cannot find the person here. I will go back
            ↪ .''
        go_to_location  ``designated_start''
        say_with_args   ``I did not find %s at the %s to guide to
            ↪  %s.''
END IF
END GRAPH
```

Listing E.19: TAIG for “Guide a Person to Somewhere, Knowing Where They are to Go But Not Where They Start” (“guidenostart”) GPSR Subtask.

```
START GRAPH
queue_subtask_statement ``planner_config/planner/subtask''
say_with_args   ``I am going to guide %s to the %s.  But I don't
    ↪ know where the person is.''
say     ``Would you tell me the location of the person?  FOr
    ↪ example, the sofa.''
begin_frame_listen      ``location''
WHILE NOT frame_heard
END LOOP
add_location [0]
say_with_args   ``Thank you!  I am going to guide %s at the %s to
    ↪  the %s.''
go_to_location  [None]
say_with_args   ``I am looking for %s now.  I am at the %s.  I
    ↪ will go to %s later.''
find_person_in_place
IF person_found
        say_with_args   ``Hello %s.  My operator told me to guide
            ↪  you from %s to the %s.''
        say     ``Would you follow me? Please answer yes or no.''
        begin_frame_listen      ``yesno''
        WHILE NOT frame_heard
        END LOOP
        IF positive_response
                say     ``OK. Please follow me.''
                go_to_location  [None]
                say     ``We have reached our destination. Thank
                    ↪ you.  I will go back now.''
                go_to_location  ``designated_start''
        ELSE
                say     ``Alright. Thanks for letting me know. I
```

```
                  ↪ will go back now.''
                go_to_location  ``designated_start''
        END IF
ELSE
        say     ``I cannot find the person here. I will go back''
        go_to_location  ``designated_start''
        say_with_args   ``I did not find %s at the %s to guide to
           ↪  %s.''
END IF
END GRAPH
```

Listing E.20: TAIG for "Meet a Person at a Beacon and Guide, not Told to Where to Guide" ("guidenoend") GPSR Subtask.

```
START GRAPH
queue_subtask_statement ``planner_config/planner/subtask''
say_with_args   ``I am going to guide %s at the %s. But I don't
   ↪ know where to guide the person.''
say     ``Would you tell me the location of the destination?  For
   ↪  example, the sofa.''
begin_frame_listen      ``location''
WHILE NOT frame_heard
END LOOP
add_location  [1]
say_with_args   ``Thank you!  I am going to guide %s at the %s to
   ↪  the %s.''
go_to_location  [None]
say_with_args   ``I am looking for %s now.  I am at the %s.  I
   ↪ will go to %s later.''
find_person_in_place
IF person_found
        say_with_args   ``Hello %s. My operator told me to guide
           ↪ you from %s to %s.''
        say     ``Would you follow me? Please answer yes or no.''
        begin_frame_listen      ``yesno''
        WHILE NOT frame_heard
        END LOOP
        IF positive_response
                say     ``Ok. Please follow me.''
                go_to_location  [None]
                say     ``We have reached our destination.  Thank
                  ↪  you!  I will go back now.''
                go_to_location  ``designated_start''
        ELSE
```

```
                    say      ``Alright. Thanks for letting me know. I
                       ↪ will go back now.''
                    go_to_location  ``designated_start''
          END IF
ELSE
          say      ``I cannot find the person here.  i will go back
             ↪ .''
          go_to_location  ``designated_start''
          say_with_args   ``I did not find %s and the %s to guide
             ↪ to %s.''
END IF
END GRAPH
```

Listing E.21: TAIG for “Take the object and deliver it to a person with a certain gesture in a room.” (“deliverobjgesture”) GPSR Subtask.

```
START GRAPH
queue_subtask_statement ``planner_config/planner/subtask''
say_with_args  ``I am going to find someone %s and deliver the %
    ↪ s at the %s.''
say     ``It does not look like something that I can pick up.
    ↪ Would you be able to help me deliver it? If you do this for
    ↪  me, my team's humans will give you candy later.  Please
    ↪ answer yes or no.''
begin_frame_listen      ``yesno''
WHILE NOT helper_response_heard
END LOOP
IF positive_response
        say     ``Ok. Please take the object and follow me.''
        go_to_location  [None]
        say_with_args   ``I am looking for a person %s to deliver
            ↪  the %s.  I am at the %s now.''
        find_any_person
        IF person_of_interest_found
                say_with_args   ``Hello!  You must be %s.  My
                    ↪ operator helped me bring the %s here at the
                    ↪  %s. Please take it from my operator.''
                say     ``The task is done.  Thank you for
                    ↪ helping me bring it over.  I will go back
                    ↪ .''
                go_to_location  ``designated_start''
        ELSE
                say_with_args   ``I cannot find anyone %s here to
                    ↪  deliver the %s to.  I am at the %s.  I
                    ↪ will go back.''
                go_to_location  ``designated_start''
        END IF
ELSE
        say_with_args   ``Sorry.  I am not able to find the
            ↪ person %s and deliver th %s at the %s, becaue I
            ↪ cannot pick it up.''
END IF
END GRAPH
```

Appendix F

List of All Primitives Used in the GPSR task

Please find listed on the following pages all the primitives used in the TAIGs noted in Appendix E and in Section 4.2.

	id	Name	Description
Action			
	add_location	Process Location from Frame	Add any beacons, placements, or rooms specified in the most recent Frame to appropriate locations in memory (destinations and say_args)
	answer_question_gpsr	Answer Frame (GPSR)	Answer a question asked by a human in the context of a GPSR subtask interaction
	approach_obj	Approach Object	Approach object in front of Pepper
	begin_frame_listen	Listen for Frame	Listen for a command or statement (subscribe to topic to which the speech module will publish frame information)
	clear_subtask_info	Clear Subtask Info	Clear values in memory after GPSR subtask is complete
	count_ppl_with_attr	Count People With Attribute	Count the number of people who have a particular attribute
	count_objcats	Count ObjCats	search the room in order to count how many of the object or category there are. if none is specified, retrieve from memory. save this inormation to memory and queue it in say_args
	find_any_person	Search For Person	Look arond until Pepper sees any human
	find_person_in_place	Find Person In Place	Search for a person by spinning in a circle until the person is found
	go_to_location	Go To Location	Go to the Location specified by the string passed as argument. If string is None, go to a location specified in memory.
	inc_counter	Increment Counter	Increment the counter variable by 1
	init_head	Initialize Head	Put head facing forward and up
	learn_name_from_frame	Learn Name From Frame	put name learned from recent frame into memroy and say_args
	load_gpsr_by_frame	Load GPSR TAIG for Subtask	For GPSR, determine the appropriate TAIG to run as a child graph based on the command frame, and note this graph in memory
	match_frame	Match Frame	Check the last speech heard (or series of speech), and determin the Frame of the question being asked / statement being made. Specify this frame on memory. last_question_frame
	point	Point At Location of Interest	Point at a specified coordinate
	put_down_arm	Put Down Arm	Pepper lowers any raised arms
	queue_subtask_statement	Queue Subtask Statement	For GPSR, parse a command frame and and add parameter information to memory
	report_human_attribute	Report Human Attribute	Save a person's attributes and room they are in to say_args in memory
	run_ig	Run Graph	[Built In] Run the specified graph, or load from memory the name of a graph and execute it as a child graph of the current graph, passing the same memory and primitive library
	sa_frame	Set Say Args for Frame	Set [say] arguments on memory from the frame in preparation for repeating the question or command in Pepper's own words
	sa_question_gpsr	Set Say Args for Frame (GPSR)	Set [say] arguments on memory from the frame in preparation for repeating the GPSR command in Pepper's own words
	say	Say	Perform Text to Speech on the Input Argument
	say_with_args	Say With Args	Perform Text to Speech on Input Argument as a template, filling in certain strings from say_args list in memory
	search_extreme_objcat	Search for Object with Property	search current room for a specified object that has the specified property (if not explicitly specified, object and attribute retrieved from memory). announce when done searching, and save information to memory

	id	Name	Description
	search_for_catobj	Search For CatObj	Search the current room for the specified thing (if object, search for the object, if a category, search for any object in that category) Pepper will go to the center of the room and spin around, or for a large room, go to certain specific points in the room and spin around in those locations in order to search
	set_answer_gpsr	Set Answer GSPR	queue say_args in memory with appropriate conversation starter that Pepper was requested to use to begin a conversation with a person during a GPSR subtask
	set_counter	Set Counter	Set the counter variable to a value
	set_helper_info	Set Helper Info	If Pepper has found a human during the most recent search, save their detected attributes to memory (i.e. shirt color) in preparation for asking them for help moving an object
	set_say_args_deliver_obj_from	Setup Say Args for Deliver Object From	in the Deliver Object From subtask, take the information about locations, person name, and object and queue them in say_args in memory to be put into say args templates when say_args is next called
Condition			
	catobj_found	Is Category or Object Found	Has a Category or Object been found in the most recent search
	counter_less	Is Counter Less Than	Check if the counter variable is less than the passed argument x
	found_objcat_extreme	Found ObjCat Extreme	Has recent search resulted in finding an object (or object in a particular category) matching the appropriate attributes, and been stored in memory?
	frame_heard	Has Frame Been Heard	Check if any frame has been detected since listening was last begun
	helper_response_heard	Helper Response Heard	Has a request for help been answered (in any manner)
	human_attr_found	Found Person of Particular Attribute	Has recent search discovered a person matching the particular attributes noted in memory?
	is_answer_question_gpsr	Is Answer Question GPSR	Is the GPSR task involving answering a question from a person?
	is_report_name	Report Name	Is Pepper to report a person's name?
	person_found	Is Person Found	has a person been found in the most recent search
	person_of_interest_found	Found Person of Interest	Has recent search populated persons_of_interest in memory (ie because they have a certain attribute)
	positive_repsonse	Positive Response	Assuming the most recent frame was either a "yes" or "no" category of response from a human, was the response "yes" /"agree" etc?

Bibliography

[1] José Luis Ambite, Craig A Knoblock, and Steven Minton. Plan optimization by plan rewriting. *Intelligent Techniques for Planning*, 2005. 2.1

[2] John R Anderson, Dan Bothell, Christian Lebiere, and Michael Matessa. An integrated theory of list memory. *Journal of Memory and Language*, 38(4): 341–380, 1998. 2.1

[3] Noriaki Ando, Shinji Kurihara, Geoffrey Biggs, Takeshi Sakamoto, Hiroyuki Nakamoto, and Tetsuo Kotoku. Software deployment infrastructure for component based rt-systems. *JRM*. 2.1

[4] Brenna D. Argall, Brett Browning, and Manuela Veloso. *Learning robot motion control with demonstration and advice-operators*, pages 399–404. 12 2008. ISBN 9781424420582. doi: 10.1109/IROS.2008.4651020. 2.1

[5] Brenna D Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. A survey of robot learning from demonstration. *Robotics and autonomous systems*, 57(5):469–483, 2009. 2.1, 2.2

[6] Jean-christophe Baillie, Akim Demaille, Quentin Hocquet, and Matthieu Nottale. Events!(reactivity in urbiscript). *First International Workshop on Domain-Specific Languages and models for Robotic systems*, 2010. 2.1

[7] JP Bandera, JA Rodriguez, L Molina-Tanco, and A Bandera. A survey of vision-based architectures for robot learning by imitation. *International Journal of Humanoid Robotics*, 9(01):1250006, 2012. 2.2

[8] Kayce Basques, Andreas Dewes, Christoph Neumann, et al. The little book of python anti-patterns. `https://github.com/quantifiedcode/python-anti-patterns`, 2015. 1

[9] Youssef Bassil. Neural network model for path-planning of robotic rover systems. *arXiv preprint arXiv:1204.0183*, 2012. 7

[10] Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. Verifiable reinforcement learning via policy extraction. In *Advances in Neural Information Processing Systems*, pages 2494–2504, 2018. 2.3.3

[11] Ned Batchelder. Eval really is dangerous, June 2012. URL `https://nedbatchelder.com/blog/201206/eval_really_is_dangerous.html`. 1

[12] Jose-Luis Blanco, Javier Gonzalez, and Juan-Antonio Fernandez-Madrigal. Consistent observation grouping for generating metric-topological maps that improves robot localization. In *Robotics and Automation, 2006. ICRA 2006.* 2.1

[13] Jason Bloomberg. Don't trust artificial intelligence? time to open the ai 'black box'. *Forbes*, 2018. 1.1, 7

[14] Tolga Bolukbasi, Kai-Wei Chang, James Y Zou, Venkatesh Saligrama, and Adam T Kalai. Man is to computer programmer as woman is to homemaker? debiasing word embeddings. In *Advances in Neural Information Processing Systems*, pages 4349–4357, 2016. 7

[15] Olivier Bonnet-Torres and Catherine Tessier. From team plan to individual plans: A petri net-based approach. In *Autonomous Agents and Multiagent Systems (AAMAS)*, 2005. ISBN 1-59593-093-0. 2.1

[16] Daniel Borrajo, Anna Roubíčková, and Ivan Serina. Progress in case-based planning. *ACM Computing Surveys*, 47(2):35, 2015. 2.1

[17] Craig Boutilier, Richard Dearden, Moises Goldszmidt, et al. Exploiting structure in policy construction. In *IJCAI*, volume 14, pages 1104–1113, 1995. 2.3.3

[18] Craig Boutilier, Richard Dearden, and Moisés Goldszmidt. Stochastic dynamic programming with factored representations. *Artificial intelligence*, 121(1-2): 49–107, 2000. 2.3.3

[19] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016. 6.1, 7

[20] Brett Browning, James Bruce, Michael Bowling, and Manuela Veloso. Stp: Skills, tactics, and plays for multi-robot control in adversarial environments. *Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering*. 2.1, 3.1

[21] Herman Bruyninckx. Open robot control software: the orocos project. In *Robotics and Automation, 2001. Proceedings 2001 ICRA.* 2.1

[22] Nicola Castaman, Elisa Tosello, and Enrico Pagello. Conditional task and motion planning through an effort-based approach. In *2018 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, pages 49–54. IEEE, 2018. 2.1

[23] David Chapman and Leslie Pack Kaelbling. Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In *IJCAI*, volume 91, pages 726–731, 1991. 2.3.3

[24] Sonia Chernova and Manuela Veloso. Learning equivalent action choices from demonstration. In *IROS 2008*, pages 1216–1221. IEEE, 2008. 2.1

[25] Francisco Cruz, Johannes Twiefel, Sven Magg, Cornelius Weber, and Stefan Wermter. Interactive reinforcement learning through speech guidance in a domestic scenario. In *2015 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2015. 2.2

[26] Heriberto Cuayáhuitl. Robot learning from verbal interaction: a brief survey. *Proceedings of the New Frontiers in Human-Robot Interaction*, 2015. 2.2

[27] Michiel de Jong, Kevin Zhang, Aaron M Roth, Travers Rhodes, Robin Schmucker, Chenghui Zhou, Sofia Ferreira, João Cartucho, and Manuela Veloso. Towards a robust interactive and learning social robot. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, pages 883–891. International Foundation for Autonomous Agents and Multiagent Systems, 2018. 4.2.2

[28] Simon DeDeo. Wrong side of the tracks: Big data and protected categories. *CoRR*, 2014. 7

[29] Coline Devin, Abhishek Gupta, Trevor Darrell, Pieter Abbeel, and Sergey Levine. Learning modular neural network policies for multi-task and multi-robot transfer. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2169–2176. IEEE, 2017. 2.1

[30] Thomas G Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000. 6.1

[31] Debidatta Dwibedi, Ishan Misra, and Martial Hebert. Cut, paste and learn: Surprisingly easy synthesis for instance detection. In *The IEEE international conference on computer vision (ICCV)*, 2017. 4.2.2

[32] Guglielmo Gemignani, Steven Klee, Daniele Nardi, and Manuela Veloso. Graph-Based Task Libraries for Robots: Generalization and Autocompletion. In *AI*IA'15*. 3.1

[33] Guglielmo Gemignani, Emanuele Bastianelli, and Daniele Nardi. Teaching robots parametrized executable plans through spoken interaction. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, pages 851–859. International Foundation for Autonomous Agents and Multiagent Systems, 2015. 2.1

[34] Ujjwal Das Gupta, Erik Talvitie, and Michael Bowling. Policy tree: Adaptive representation for policy gradient. In *AAAI*, pages 2547–2553, 2015. 2.3.3

[35] Larry Hardesty. Making computers explain themselves. *MIT News*, 27:2016,

2016. 7

[36] Todd Hester, Michael Quinlan, and Peter Stone. Generalized model learning for reinforcement learning on a humanoid robot. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 2369–2374. IEEE, 2010. 2.3.3

[37] Jaitus Hihn and Tim Menzies. Data mining methods and cost estimation models: Why is it so hard to infuse new ideas? In *Automated Software Engineering Workshop (ASEW), 2015 30th IEEE/ACM International Conference on*, pages 5–9. IEEE, 2015. 1.1

[38] Yunyi Jia, Lanbo She, Yu Cheng, Jiatong Bao, Joyce Y Chai, and Ning Xi. Program robots manufacturing tasks by natural language instructions. In *2016 IEEE International Conference on Automation Science and Engineering (CASE)*, pages 633–638. IEEE, 2016. 2.2

[39] Chen Jin, Luo De-Lin, and Mu Fen-Xiang. An improved id3 decision tree algorithm. In *2009 4th International Conference on Computer Science & Education*, pages 127–130. IEEE, 2009. 14

[40] Girish Joshi and Girish Chowdhary. Cross-domain transfer in reinforcement learning using target apprentice. *arXiv preprint arXiv:1801.06920*, 2018. 2.1

[41] Sylvain Joyeux and Jan Albiez. Robot development: from components to systems. In *6th National Conference on Control Architectures of Robots*, pages 15–p, 2011. 2.1

[42] Shinpei Kato, Eijiro Takeuchi, Yoshio Ishiguro, Yoshiki Ninomiya, Kazuya Takeda, and Tsuyoshi Hamada. An open approach to autonomous vehicles. *IEEE Micro*, 35(6):60–68, 2015. 2.1

[43] Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks. In *International Conference on Computer Aided Verification*, pages 97–117. Springer, 2017. 1.1

[44] Jaehun Kim, Julián Urbano, Cynthia Liem, and Alan Hanjalic. One deep music representation to rule them all: A comparative analysis of different representation learning strategies. *Neural Computing and Applications*, Mar 2019. ISSN 1433-3058. doi: 10.1007/s00521-019-04076-1. 2.1

[45] James Kirk, Aaron Mininger, and John Laird. Learning task goals interactively with visual demonstrations. *Biologically Inspired Cognitive Architectures*, 18:1–8, 2016. 2.2

[46] Steven D Klee, Guglielmo Gemignani, Daniele Nardi, and Manuela Veloso. Multi-robot task acquisition through sparse coordination. In *IROS 2015*. 2.1, 3.1

[47] Steven D Klee, Guglielmo Gemignani, Daniele Nardi, and Manuela Veloso. Graph-based task libraries for robots: Generalization and autocompletion. In *Congress of the Italian Association for Artificial Intelligence*, pages 397–409. Springer, 2015. 1.1, 5.1.1

[48] Will Knight. The dark secret at the heart of ai: no one really knows how the most advanced algorithms do what they do-that could be a problem. *MIT Technology Review*, 2017. 1.1

[49] Nathan Koenig and Maja J Matarić. Robot life-long task learning from human demonstrations: a bayesian approach. *Autonomous Robots*, 41(5):1173–1188, 2017. 2.2

[50] John E Laird, Kevin Gluck, John Anderson, Kenneth D Forbus, Odest Chadwicke Jenkins, Christian Lebiere, Dario Salvucci, Matthias Scheutz, Andrea Thomaz, Greg Trafton, et al. Interactive task learning. *IEEE Intelligent Systems*, 32(4): 6–21, 2017. 2.2

[51] Jangwon Lee. A survey of robot learning from demonstrations for human-robot collaboration. *CoRR*, 2017. 2.2

[52] Tania Lombrozo. Simplicity and probability in causal explanation. *Cognitive psychology*, 55(3):232–257, 2007. 2.3.2

[53] Félix López and Víctor Romero. *Mastering Python Regular Expressions*. Packt Publishing Ltd, 2014. 5.1.1

[54] Ying Lu, Liming Chen, and Alexandre Saidi. Optimal transport for deep joint transfer learning. *arXiv*, 2017. 2.1

[55] Björn Lütjens, Michael Everett, and Jonathan P How. Safe reinforcement learning with model uncertainty estimates. *CoRR*, 2018. 7

[56] Luis Manso, Pilar Bachiller, Pablo Bustos, Pedro Núnez, Ramón Cintas, and Luis Calderita. Robocomp: a tool-based robotics framework. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*. 2.1

[57] Andrew Kachites McCallum and Dana Ballard. *Reinforcement learning with selective perception and hidden state*. PhD thesis, University of Rochester. Dept. of Computer Science, 1996. 2.3.3

[58] Cetin Mericli, Steven D. Klee, Jack Paparian, and Manuela Veloso. An interactive approach for situated task specification through verbal instructions. In *AAMAS*, 2014. 2.2, 3, 3.1, 5

[59] Giorgio Metta, Paul Fitzpatrick, and Lorenzo Natale. Yarp: yet another robot platform. *International Journal of Advanced Robotic Systems*, 3(1):8, 2006. 2.1

[60] Donald Michie. Inductive rule generation in the context of the fifth generation.

In *Machine Learning Workshop*, page 65, 1983. 2.3.2

[61] Donald Michie. Current developments in expert systems. In *Proceedings of the Second Australian Conference on Applications of expert systems*, pages 137–156. Addison-Wesley Longman Publishing Co., Inc., 1987. 2.3.2

[62] Tom M Mitchell and Sebastian B Thrun. Explanation-based neural network learning for robot control. In *Advances in neural information processing systems*, pages 287–294, 1993. 1

[63] Shiwali Mohan and John E Laird. Learning goal-oriented hierarchical tasks from situated interactive instruction. In *AAAI*, pages 387–394, 2014. 2.1

[64] Vahid Mokhtari, Luís Seabra Lopes, and Armando J Pinho. Experience-based robot task learning and planning with goal inference. In *Twenty-Sixth International Conference on Automated Planning and Scheduling*, 2016. 2.2

[65] Vahid Mokhtari, Luís Seabra Lopes, and Armando J Pinho. Learning robot tasks with loops from experiences to enhance robot adaptability. *Pattern Recognition Letters*, 99:57–66, 2017. 2.2

[66] Edward J Mulrow. The visual display of quantitative information, 2002. 2.3.2

[67] MA Viraj J Muthugala and AG Buddhika P Jayasekara. Mirob: An intelligent service robot that learns from interactive discussions while handling uncertain information in user instructions. In *2016 Moratuwa Engineering Research Conference (MERCon)*, pages 397–402. IEEE, 2016. 2.2

[68] Paul Michael Newman. Moos-mission orientated operating suite. 2008. 2.1

[69] Hung Nguyen, Vu Tran, Tung Nguyen, Matthew Garratt, Kathryn Kasmarik, Michael Barlow, Sreenatha Anavatti, and Hussein Abbass. Apprenticeship bootstrapping via deep learning with a safety net for uav-ugv interaction. *CoRR*, 2018. 7

[70] Monica N Nicolescu and Maja J Mataric. Natural methods for robot task learning: Instructive demonstrations, generalization and practice. In *AAMAS*. 2.1

[71] Hae Won Park and Ayanna M Howard. Retrieving experience: Interactive instance-based learning methods for building robot companions. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6140–6145. IEEE, 2015. 2.2

[72] Pieter Philippaerts, Yves Younan, Stijn Muylle, Frank Piessens, Sven Lachmund, and Thomas Walter. Cpm: Masking code pointers to prevent code injection attacks. *TISSEC*, 2013. 1

[73] Larry D Pyeatt. Reinforcement learning with decision trees. In *Applied Informatics*, pages 26–31, 2003. 2.3.3

[74] Larry D Pyeatt, Adele E Howe, et al. Decision tree function approximation in reinforcement learning. In *Proceedings of the third international symposium on adaptive systems: evolutionary computation and probabilistic graphical models*, volume 2, pages 70–77. Cuba, 2001. 2.3.3

[75] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009. 2.1

[76] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986. 2.3.2

[77] J. Ross Quinlan. Simplifying decision trees. *International journal of man-machine studies*, 27(3):221–234, 1987. 2.3.2

[78] Aaron M. Roth, Umang Bhatt, Tamara Amin, Afsaneh Doryab, Fei Fang, and Manuela Veloso. The impact of humanoid affect expression on human behavior in game-theoretic setting. In *Proceedings of the IJCAI'18 Workshop on Humanizing AI (HAI), the 28th International Joint Conference on Artificial Intelligence*, Stockholm, Sweden, July 2018. 4.3

[79] Aaron M. Roth, Samantha Reig, Umang Bhatt, Jonathan Shulgach, Tamara Amin, Afsaneh Doryab, Fei Fang, and Manuela Veloso. A robots expressive language affects human strategy and perceptions in a competitive game. In *Proceedings of the 28th IEEE International Conference on Robot Human Interactive Communication*, New Delhi, India, October 2019. 4.3

[80] Aaron M. Roth, Nicholay Topin, Pooyan Jamshidi, and Manuela Veloso. Conservative q-improvement: Reinforcement learning for an interpretable decision-tree policy. In *arXiv 1907.01180*, 2019. 7.1, 1, 7.3

[81] Vivian S Silva, André Freitas, and Siegfried Handschuh. On the semantic interpretability of artificial intelligence models. *arXiv preprint arXiv:1907.04105*, 2019. 1.1

[82] Siddharth Srivastava, Nishant Desai, Richard Freedman, and Shlomo Zilberstein. An anytime algorithm for task and motion mdps. *arXiv preprint arXiv:1802.05835*, 2018. 2.1

[83] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018. 2.3.1

[84] Ryutaro Tanno, Kai Arulkumaran, Daniel C Alexander, Antonio Criminisi, and Aditya Nori. Adaptive neural trees. *CoRR*, 2018. 2.3.2

[85] Jesse Thomason, Shiqi Zhang, Raymond J Mooney, and Peter Stone. Learning to interpret natural language commands through human-robot dialog. In *Twenty-*

Fourth International Joint Conference on Artificial Intelligence, 2015. 2.2

[86] William T Uther. Tree-based hierarchical reinforcement learning. Technical report, Carnegie Mellon, Univ Pittsburgh, PA, Dept of Computer Science, 2002. 2.3.3

[87] William TB Uther and Manuela M Veloso. Tree based discretization for continuous state space reinforcement learning. In *Aaai/iaai*, pages 769–774, 1998. 2.3.3

[88] William TB Uther and Manuela M Veloso. The lumberjack algorithm for learning linked decision forests. In *International Symposium on Abstraction, Reformulation, and Approximation*, pages 219–232. Springer, 2000. 2.3.3

[89] William TB Uther and Manuela M Veloso. Ttree: Tree-based state generalization with temporally abstract actions. In *Adaptive agents and multi-agent systems*, pages 260–290. Springer, 2003. 2.3.3

[90] Manuela M. Veloso and Jaime G. Carbonell. Derivational analogy in prodigy: Automating case acquisition, storage, and utilization. *Machine Learning*, 10: 249–278, 1993. 2.1

[91] Huijun Wu, Chen Wang, Jie Yin, Kai Lu, and Liming Zhu. Interpreting shared deep learning models via explicable boundary trees. *CoRR*, 2017. 1.1

[92] Min Wu, Atsushi Yamashita, and Hajime Asama. Rule abstraction and transfer in reinforcement learning by decision tree. In *System Integration (SII), 2012 IEEE/SICE International Symposium on*, pages 529–534. IEEE, 2012. 2.3.3

[93] Quanshi Zhang, Yu Yang, Ying Nian Wu, and Song-Chun Zhu. Interpreting cnns via decision trees. *CoRR*, 2018. 2.3.2

[94] Qiangfu Zhao. Evolutionary design of neural network tree-integration of decision tree, neural network and ga. In *Evolutionary Computation, 2001. Proceedings of the 2001 Congress on*, volume 1, pages 240–244. IEEE, 2001. 2.3.2