LEARNING CONTIGUITY-BASED HIERARCHICAL TASK MODELS FROM DEMONSTRATION

A Thesis
Presented to
The Academic Faculty

by

Leo Thomas Rossignac-Milon

In Partial Fulfillment
of the Requirements for the Degree
Bachelors of Science in the
School of Computer Science

Georgia Institute of Technology
Dec 2014

Approved by:

Dr. Andrea Thomaz, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Henrik Christensen
School of Computer Science
Georgia Institute of Technology

Date Approved: 12/11/2014

ACKNOWLEDGEMENTS

I wish to thank my mentor, Andrea Thomaz, for peaking my interest in the field of artificial intelligence and for guiding my research through out my undergraduate career.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

TEM	Task Execution Model
HTN	Hierarchical Task Network
U-Group	Unordered Group
S-Group	Sequential Group
R-Group	Reversible Group

SUMMARY

We propose an incremental approach for learning a hierarchical task model from a series of demonstrations, where each demonstration is a permutation of a fixed number of different actions. Our hierarchical Task Execution Model, called *TEM*, is a tree, where each leaf represents an action and each node represents a composite action (or subtask). We distinguish three types of composite nodes (s-group: sequential, r-group: reversible, and u-group: unordered). Although the sub-task children of a node must always be executed as a contiguous (uninterrupted) sequence, the valid orders for that sequence depend on the node type. Hence, a TEM captures a well-defined set of contiguity and ordering constraints.

TEM may be used to test quickly whether a candidate plan of actions is compatible with the task model and also to provide a list of valid actions at any step during the lazy execution of a task. Furthermore, it may be used to explain the task model by providing (to the human) its hierarchical decomposition into subtasks and defining explicit relations (constraints) of order and contiguity amongst them. Our target application is to allow a robot to learn a task, such as a house chore, in an unsupervised manner from a series of demonstrations performed by humans. We propose an incremental learning algorithm, called TEM++, which takes as input the current TEM learned from previous demonstrations as well as a new demonstration, and which produces a new TEM.

CHAPTER 1

INTRODUCTION

An important and growing trend in robotics is focused on programming robots by demonstration. Such an approach is particularly well suited in robotics applications in the home and in flexible manufacturing cells where the same robot may need to be reprogrammed often to learn new tasks. We define the task as a sequence of actions. For example, in a home application, the task may be to cook chicken and the actions may be to open the cabinet, take out the pan, place the pan on the burner, turn on the burner, place the chicken in the pan, and so on.

In this paper, we restrict our attention to situations where the task comprises a sequence of n distinct actions. As input, we assume a series of demonstrations, where each demonstration has a one-to-one mapping between its actions and the original n actions. Our objective is to compute a hierarchical task model, which represents all groups of actions that have been contiguous in every demonstration in the series.

We nest these groups to create a hierarchical Task Execution Model, called *TEM*, which is represented by a tree. Each leaf represents an action and each node represents a composite action (or subtask) defined in terms of a group of children (nodes or leaves).

We distinguish three types of composite nodes: s-group = sequential (fixed order), r-group = reversible (reversible order), and u-group: unordered. Although the sub-task children of a node must always be executed as a continuous sequence, the valid orders for that sequence depend on the node type.

The specific contributions presented here include:

- A formal definition of the target knowledge (all groups of actions that are always contiguous throughout a series of demonstrations).
- A compact representation of this knowledge which may be used for several purposes:
    - Check whether a given action plan respects all the contiguity constraints.
    - Provide the robot with a list of valid actions, during the lazy execution of a task.
    - Provide a human teacher with a visually compact representation of what the robot understands as being the set of all the valid sequences of actions for a given task.
- An incremental algorithm, TEM++, that builds a TEM for the first demo and then updates the TEM by incorporating the information provided by each subsequent demo, one demo at a time.

CHAPTER 2

LITERATURE REVIEW

Throughout the past decade, remote robot laboratories have appeared around the globe such as the PR2 Remote Lab at the Bosch Research and Technology Center and the Telerobot of the University of Western Australia (UWA). These remote labs have mostly been used for collaboration between researchers. Robotics equipment is expensive and many smaller labs that would otherwise be unable to test their developments are given the opportunity to become more significant contributors to the field of robotics. Remote robotics labs also provide standardization allowing researchers to compare various methodologies and algorithms on a shared platform [1].

However, remote labs have a range of applications that extend beyond sharing resources. The power of the Internet can be harnessed to increase not only the number of researchers working on a robot, but to also increase public exposure to robotics. So far, publicly accessible robot controllers have mainly been used for entertainment or educational purposes. The TeleGarden at the University of Southern California allows public users to water or plant flowers in a garden remotely by controlling a robot, and the Telerobot of UWA lets anyone with internet access play with colored blocks on one of their robots [1]. On the other hand, the REAL (Remotely Accessible Laboratory) in Brazil takes the more educational route, allowing students of collaborating universities to log into their XR4000 autonomous mobile robot [2].

We believe that with the right public interface, web users could become useful subjects in experiments. By expanding the pool of subjects and decreasing the effort

involved in participation, the number and diversity of subjects involved in experiments should both rise significantly. Creating a remote lab involves 3D visualization of objects, controlling the robot easily and quickly, data logging, and robot monitoring [1]. Since our remote lab would be focused on Learning from Demonstration (LfD), the development involves designing a LfD algorithm that is appropriate to the context.

Learning from Demonstration, a subset of Supervised Learning, consists of creating task models or policies by using examples of the task. There are three main groups of methodologies used today to learn these task models. The first is to build a mapping function that allows the robot to calculate the probability of performing each action based on its observations of the current world state. Alternatively, the robot can create a system model. This model tries to understand how each action affects the world state and is often combined with a reward function to help the robot choose which action will lead it to a more preferable state. The third method is to create preconditions and post-conditions for each action. These conditions can be used to plan the task [3].

On the data collection side of LfD, there are two ways to make demonstrations: imitation and demonstration. In imitation, the teacher performs the task, and the robot then tries to map the teacher's actions to his own. In demonstration, the teacher physically moves the robot. For each of these methods, sensors can be placed either on the teacher or on the robot, and resulting mappings sometimes need to be performed [3]. However, remote robotics involves teleoperation, an ideal type of demonstration. In teleoperation, there is a direct mapping for the sensors and for the actions since both sets of data are gathered from the robot's motor and sensory systems. All three policy-learning methods described above can be used successfully using teleoperation. However,

one of the goals of our remote robotics laboratory is for the users to be able to better understand the current task model as they provide further demonstrations to help refine the model. For this visualization to be useful for more complex tasks, we must provide a mixture of ordering of actions as well as provide a hierarchical structure to the actions. These requirements, will constrain the methodologies that we can viably use.

Many researchers have attempted to solve partial ordering and hierarchical task structure with the use of structured dialogue from the teacher. A group of professors from Carnegie Mellon University's School of Computer Science have developed a system of learning hierarchical task structure using verbal commands. The teacher can create and name tasks as well as describe the parameters involved in the task. Teachers can then reuse premade tasks within more complex tasks. This system, however, has the disadvantage that the teacher must be familiar with the very structured set of verbal commands allowed. Tasks must also be verbally programmed by using 'if', 'then', and 'go to' statements. Moreover, the task models are stored as finite state machines, which do not allow the robot to easily update them as new knowledge is acquired [4].

Researchers at Mitsubishi Electric Research Laboratories also used structured dialogue, but their goal was to allow task models to incorporate partial ordering constraints. Using their system, the teacher can annotate tasks by writing out all preconditions for every action. The robot can then combine these annotations with a physical task demonstration to create the generalized task model. While this system allows for quick learning, it relies on the teacher to be familiar with the complex task programming language [5]. Neither of the methods that use structured dialogue is suitable for LfD, as the motive behind LfD is that end-users should be able to teach robots tasks

with little to no training. Also, for very complex tasks, even an expert may not be able to enumerate all conditions or partial ordering constraints. It seems reasonable that the teacher should simply be responsible for providing demonstrations and answering questions.

Fortunately, methods for learning partial task ordering without the use of structured dialogue have also been researched. A generalized task model can be formed using state related preconditions for each action combined with the ability to detect repetitions within a demonstration. A repetition is limited to a sequence of actions that appears multiple times in the demonstration. This type of model allows the robot to calculate which action is appropriate for its current state as well as to understand which sequences of actions can be performed repeatedly until the precondition is no longer present [6]. For example, a robot could learn that when it sees blocks inside a box, it should remove all blocks from the box until no more blocks are left in the box. Alternatively, it is also possible for task preconditions to be based on previous actions instead of on the current world state. One way to create such a task model is to find the longest common action subsequence of multiple demonstrations and then append any alternate action paths to this base model. The end result is an action plan that shows all valid possible paths from which the robot can choose. This technique allows for the model to be updated as more demonstrations are incorporated [7]. Using this model, a robot could for example learn that it can only place a ball in a box after having removed the block from it, which can in turn only happen after having opened the box. Unfortunately, both of these techniques are only for flat representations, making the current implementations incompatible for hierarchical tasks.

However, action preconditions and world state preconditions are not mutually exclusive. The two can be combined for an even better task model. Algorithms that create these types of task models have been implemented at the University of South Carolina, with action preconditions taking precedence over world state preconditions. Perhaps more importantly, the behaviors in the task model were abstracted to hold either a single action or a sequence of sub-behaviors. If a behavior's action preconditions and world state precondition are both met, the behavior is triggered, which in turn triggers sub-behavior. When the behavior's sub-behaviors have all finished, the behavior itself signals that it has been completed and the task moves to a new behavior. Detecting that a primitive behavior is completed involves checking that the action's post-conditions are now a part of the world state. This implementation allows for both partial ordering constraints and hierarchical structure. The ability for behaviors to be abstracted with as many levels as needed is a powerful and much needed tool in LfD. While this task model structure is very promising, the model was preprogramed into the robot. The model was not learned from demonstration [8].

Existing algorithms that learn Hierarchical Task Networks (HTNs) by observation require the demonstration set to be coupled with either pre-encoded concepts or pre-encoded subtasks. For example, an algorithm developed at Stanford's Computational Learning Laboratory uses concepts that "are encoded in a hierarchical language and shape the system's beliefs about the domain." The system then uses back-wards chaining to find sequences of actions that resulted in a change in one of the domain concepts. These sequences are then labeled as sub-tasks [9]. In more recent work, the HTN-MAKER algorithm detects tasks in a solution by finding intervals in the tasks where the

initial state and final state respectively match the pre-conditions and goals of the pre-encoded task. HTN-Maker also requires all primitive actions to be pre-encoded in a similar manner [10]. Both of these HTN learning algorithms are limited to totally ordered sequences of subtasks.

In order to create a remote robotics laboratory where the robot is being taught in real time by untrained public internet users, the LfD algorithm used must be able to create task models that take into account both the ordering of behaviors as well as the hierarchical structure within behaviors. Moreover, the learning algorithm must be able to update its model on the fly as new demonstrations are provided. This algorithm must not be based on structured dialogue; nor must it force the users to pre-program or pre-annotate tasks or concepts. Developing a task model that satisfies these requirements will achieve a secondary purpose by allowing it to be visualized in a manner that is more intuitive for the teacher. According to Russell, Halit Bener, and Chernova, end users acting as teachers for robots have a desire to understand what the robot is thinking or what the robot already knows about the task [11]. By showing teachers the current model in an intuitive manner, they will be able to understand what partial ordering option or hierarchical division the model has yet to learn. Thus, the user will be able to provide demonstrations that best refine the model, maximizing the value of each demonstration and increasing the learning curve of the robot.

CHAPTER 3

TEM AND STEM: TERMINOLOGY, NOTATION, AND CONSTRUCTION

In this section, we define our terminology, introduce a notation, and propose a construction process for computing a TEM from a given series of sequence demonstrations.

Groups

We associate each action with a ***label***. Each label must have a unique identifier. For readability, we will let the identifiers be single lowercase characters for actions and single uppercase characters for composite actions.

Therefore, we may denote each demonstration by a string, such as "axdfg". Because we assume that all actions are different, for simplicity and without loss of generality, we assign these characters so that the string of the first demo is always "abcde…".

Let *n* denote the number of actions.

For example, when n=9, the input may be:

Series 1

Demo 1: "abcdefghijkl"

Demo 2: "bdacefgkljih"

Demo 3: "kljihabdcefg"

Throughout this paper, we use the term ***group*** to denote two or more labels that appear contiguously in every demonstration. A single demo of *n* labels (actions) defines n(n+1)/2-n groups. For example, a single demonstration of "uxv" defines the 3*4/2 -3 = 3 groups: ux, xv, and uxv. The groups of a series are found in the intersection of the groups of all the individual demos. In Series 1, "abcd" and "hi" are both groups. Clearly, the entire string "abcdefghijkl" is also a group.

A sorted group (abbreviated ***s-group***) is a group of labels that appear in the same exact order in all demos. Additionally, an s-group should not be contained, as a proper subset, in any other s-group. For example, in Series 1, "efg" is an s-group, but "ef" is not, because "ef" is contained in "efg".

A reversible group (abbreviated ***r-group***) is a group of labels that always appear either in a given order or in the exact reverse order. Again, an r-group is not contained, as a proper subset, in any other r-group. For example, in Series 1, "hij" is a R-Group.

An unordered group (abbreviated ***u-group***) is a group of size greater than 2 that does not contain a group as a proper subset. For example, in Series 1, "abcd" is a u-group, but "hijkl" is not a u-group, because "hijkl" contains the groups "hij" and "kl". Observe that all substrings of an s-group and of an r-group are also groups of the series.

For convenience, we use the term ***TEM-group*** to refer to any of the s, r, or u groups. Note that the TEM-groups of a series are a subset of all the groups of that series. We use the following notation to identify TEM-groups in our textual representation of a TEM.

The labels in a u-group are delimited by hard brackets: '[' and ']'.

The labels in an s-group are delimited by angle brackets: '<' and '>'.

The labels in an r-group are delimited by curly brackets: '{' and '}'.

For example, in Series 1, we identify the following TEM-groups: the unordered group [abcd], the sequence <efg>, the reversible group{hij}, and the sequence <kl>.

TEM

In Series 1, the group "hijkl" is neither a u-group, nor an s-group, nor a r-group. However, "hijkl" is a group of Series 1, since these 5 actions occur contiguously in all demos. To capture such knowledge, we define our Task Execution Model, or ***TEM***, as a nested (hierarchical) formulation of the groups.

To do so, we treat lowest-level TEM-groups as ***composite actions*** and assign a different label (upper case character) to each one of them.

For example after identifying the TEM-groups in Series 1, we can re-rewrite its demos as:

Series 1

Demo 1: "XYZW"

Demo 2: "WZXY"

Demo 3: "ZWXY"

Where:
X = [abcd]
Y = <efg>
Z = {hij}
W= <kl>

U-Group: [unordered]
S-Group: <sequential>
R-Group: {reversible}

Then, we perform the same TEM-group extraction on this new formulation. We repeat this process recursively until a single TEM-group is produced.

From the re-rewritten demonstrations above, we find a new s-group "XY" and a new r-group "ZW".

Now, have a partially constructed TEM with two TEM-groups <XY> and {ZW} which we can expand as < [abcd] <efg> > and { {hij} <kl> }.

Before creating the final TEM, we must correct a flaw in the current construction process. Notice that the r-group { {hij} <kl> } exactly encodes the following 5 groups: "hi", "ij", "hij", "kl" and "hijkl". However, the groups "jkl" and "ijkl" are always present in the three demonstrations. The r-group {hij} and its parent r-group {{hij} W} only appear in a reversed order when the other also does. Due to this fact, the r-group {hij} does not merit to be an r-group of its own. Thus, the parent r-group must absorb the child r-group, yielding {hij<kl>}.

More generally, we add a step in the construction process as follows:

> Absorption
>
> When creating an r-group or an s-group, p, we must check all of its children.
>
> If one of the children, c, is also an r-group or an s-group (respectively) and c has always appeared in the same order as p, then p must absorb c.
>
> Absorbing c requires p to remove its child pointer to c and replace it with pointers to all of s's children.

After applying this rule, we modify the partially constructed TEM of Series 1 from: < [abcd] <efg> > { {hij} <kl> } to <[abcd]efg> {hij<kl>}. Notice that the s-group <efg> has also been absorbed.

Let us now complete the TEM by performing another iteration of TEM-group extraction.

Series 1

Demo 1: "UV"

Demo 2: "VU"

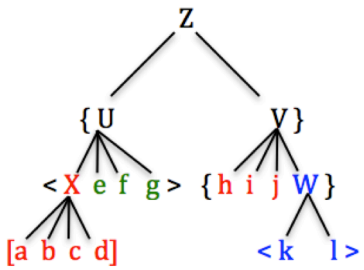
Demo 3: "VU"

Where:
U = <[abcd]efg>
V = {hij<kl>}

U-Group: [unordered]
S-Group: <sequential>
R-Group: {reversible}

We see that the two TEM-groups are reversible, and so we create a final r-group {UV}. Since all demonstrations have been resolved to a single r-group, the construction process is complete.

Figure 3.1



The final TEM of Series 1 is:

{UV} = { <[X]efg> {hij<W>} } = { <[abcd]efg> {hij<kl>} }.

This TEM is displayed in its graphical form in Figure 3.1.

Notice that V={Z<W>} is an r-group child of the {UV} r-group. However, V should not be absorbed, since in Demo 3, V appears in order while UV appears in a reversed order: {Z<W>}U.

Since the construction proposed above considers all the demos simultaneously and does not take their order into account, the TEM produced is independent of the order in which the demos are listed. Moreover, since we remove unnecessary nested s-groups, all equivalent TEMs have a unique structure. Two TEMs are **equivalent** if they encode the same groups.

Legal Traversal

Consider a tree representation of a TEM. The leaves represent the original actions. The composite (non-leaf) nodes represent composite actions and are each associated with a type: s, r or u group.

A ***legal traversal*** of a node visits all its children in some order, which depends on the type of the node. An s-group visits its children in the left-to-right order, while; an r-group visits them in either the left-to-right or the right-to-left order. A u-group visits them in any order.

To extract all the valid sequences encoded in a TEM, we enumerate all possible combinations of legal traversals and output the corresponding sequence. For example, the TEM { {ab} <cd> } produces valid permutations: "abcd", "bacd", "cdab", and "cdba".

U-Group: [unordered]
S-Group: <sequential>
R-Group: {reversible}

Lazy Legal Traversal

A TEM can be traversed lazily, outputting all legal candidate actions at any point during the traversal. This is accomplished by keeping track of which nodes have already been completed as well as the current node, N, which is initially the root node. To return a list of valid candidate actions, we recursively visit the children of N that could be legally traversed next, adding all encountered leaf nodes to the list. When an action is chosen, we mark it as complete, and set N to the selected action's parent. If all of N's children have been completed, we mark it as complete, and ascend the tree.

Minimum Number of Demos

Any TEM may be learned from only three carefully chosen demos.

For example, the TEM: {[abc]<defg>{h<jk>l}} can be learned from Series 2.

Series 2

U-Group: [unordered]
S-Group: <sequential>
R-Group: {reversible}

Demo 1: "abcdefghjkl"

Demo 2: "hjkldefgbac"

Demo 3: "ljkhdefgbca"

To prove this assertion, first observe that learning happens in parallel at all levels. Then consider that only one demo is needed to define the valid permutation of any s-group. Two demos suffice to define the two permutations of a single r-group. However, as shown above, a third demo is needed to create an r-group that has a child r-group. Finally, a u-group may also require a third demo. A third demo is only required for u-groups with only 3 children. For example [abc] can be learned from "abc", "cba", and "acb", while [abcd] can be learned from "abcd" and "bdac". Note that [abcd...wxyz] can be learned from "abcd…wxyz" and "…wcyazbxd…".

Adding Preconditions

A TEM can be combined with action preconditions to create a more robust task model. Of course, s-groups and r-groups already have an exact ordering for their children, so no additional information is needed to capture the preconditions within them. However, we can use action preconditions to add ordering constraints to the children of a u-group.

Testing if composite action C_1 of a u-group is a precondition of a composite action C_2 in that same u-group is simple. Due to the contiguity of each of the two children, we can arbitrarily chose a descendent d_1 of C_1 and a descendent d_2 of C_2 and test if one is a precondition of the other.

The traversal of a u-group can easily be modified to respect preconditions. Of course, when using a TEM along with preconditions, more than 3 demos may be needed to specify the complete task model.

CHAPTER 4

INCREMENTAL CONSTRUCTION OF TEM, TEM++

In this section, we propose an incremental process, TEM++.

If we are given the TEM T_i that encodes the groups of the first i demos and we are given the next demo D_{i+1}, we can incrementally compute the TEM T_{i+1}.

U-Group: [unordered]
S-Group: <sequential>
R-Group: {reversible}

The Incremental Approach

Our incremental process transforms D_{i+1} into T_{i+1} by processing the composite nodes of S_i into D_{i+1}, in a depth-first order, and possibly rearranging their children into new composite nodes.

Example 1:

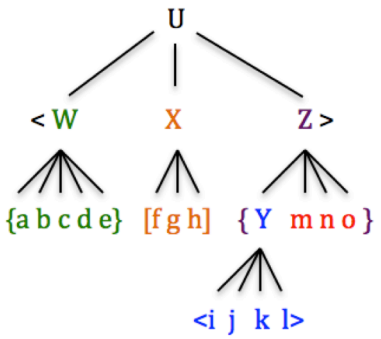
Starting with T_i= < {abcde} [fgh] {<ijkl>mno} >

and D_{i+1}= "onklmjifhagbcde",

we obtain T_{i+1}={ { on [<kl>m{ji}] } [fhag{e{bcd}}] }

Notice that the graphical version of T_i is shown in Figure 4.1.

Figure 4.1
T_i



We explain below how a single composite node N of T_i is processed into D_{i+1}.

Pieces:

When we process N into D_{i+1}, we attempt to form a single new TEM-group out the children of N. However, if the children of N do not appear contiguously, this is not possible. Instead, we create several new TEM-groups. We re-label these TEM-groups (or ungrouped individual children) with a piece label. A piece label has the same unique

identifier as N's label. In addition, a piece label records the total number of pieces that also came from N, its brothers.

If N is a U-Group:

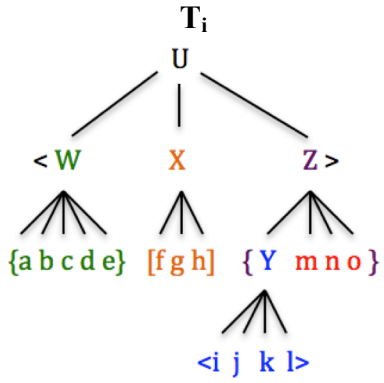
By definition of a u-group, we know that no subset of these N's children has always been contiguous. In Example 1, [def] is an unordered group. Even though "de" appears contiguously in the demo, it does not make sense to form a new TEM-group out of "de" in the next T_{i+1}. Since "de" is in an unordered group, there must have been some past demonstration where "d" and "e" did not appear contiguously. Otherwise, "de" would have been its own TEM-group.

U-Group: [unordered]
S-Group: <sequential>
R-Group: {reversible}

In order to process a u-group into a demo we must:

1. Find all the children c of N (or all the pieces of c is c was broken when it was processed)
2. If all c are contiguous in D_{i+1}
    a. Create u-group with the same label as N which contains all c as children
    b. Assign a new label with a unique id to any c which currently has a piece-label
3. Otherwise, re-label all c as pieces of N

X^3 indicates a piece of X from a brotherhood of size 3.



Example 1 (after processing [fgh]):

D_{i+1}= "onklmjifhagbcde" becomes

D_{i+1}= "onklmji$X^3X^3aX^3$ebcd"

If N is a S-Group or R-Group:

There are three main parts in this process. Each part triggers the next, in a cyclical manner.

1 - Find: The goal of this step is to find all groups of children of N that are contiguous both in the TEM-group and in the demo. Again, note that in the demo some of Ns children may have been split into pieces when they were previously processed. No piece can be in a group if that group does not also contain all of its brothers.

In order for the next two parts to work correctly, we find groups from smallest to largest size.

Notice that Find is a good candidate for dynamic programming.

2 - Validate: Whenever a contiguous group is found in Find, we must ensure that it merits becoming a new TEM-group in the next TEM. A found group will only become a new TEM-group if it does not split apart an existing TEM-group created in this iteration. The newly created TEM-group becomes a specific type of TEM-group as defined below:

N is a…	**The group contains pieces**	**The group does not contain pieces**		
		The order of the group in D is the _____ as it is in N		
		same	**reverse**	**other**
S-Group	u-group	s-group	r-group	u-group
R-Group	u-group	r-group	r-group	u-group

As described in the Absorption rule of the TEM construction, we want to ensure that no faulty r-groups or unneeded s-groups are produced: no r-group or s-group, p, should contain a child group whose order is always the same as p's order.

When constructing a TEM incrementally, determining if a parent should absorb a child is straightforward. When we create an s-group or an r-group, p, we must absorb any of its children, c, which satisfy the following two properties:

- c has also been created during the processing of N
- In D, the children of c appear in the same relative order as the children of p

Since p and c both come from the same ordered group N, their children are guaranteed to have always been in the same relative order in all past demonstrations.

On the other hand, if two groups were not both created during the processing of the N, we can be certain that the relative ordering of their children has not always been the same, by the structural properties that absorption guarantees for the previous TEM. Therefore, there is no need to consider abortion of two groups not created during the same processing.

3 - Label: Once a new TEM-group has been validated and created, we re-label it with a unique id. Moreover, any of its children that are labeled as pieces must also be relabeled so they longer appear as pieces.

Once the Find task reports that no larger groups can be found, we must label the topmost TEM-groups created during N's processing. If N's children appear contiguously in the demo, there will only be a single topmost TEM-group. In this case, the topmost TEM-

group copies its label from N. Otherwise, there will be multiple top-most TEM-groups, all of which are re-labeled as pieces of N.

U-Group: [unordered]
S-Group: <sequential>
R-Group: {reversible}

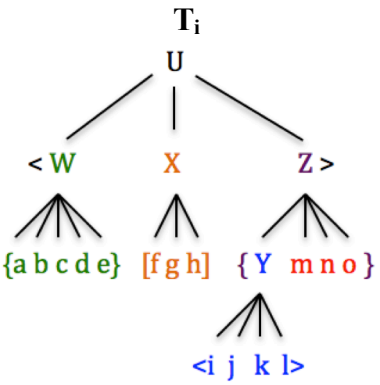
Example 1 (after processing {abcde}):

D_{i+1}= "onklmjifhagbcde"

= "onklmji$X^3X^3aX^3$ebcd" becomes

D_{i+1}= "onklmjifhag{e{bcd}}"

= "onklmji$X^3X^3W^2X^3W^2$"



First we must find the smallest contiguous groups. For simplicity, let us assume we find these groups from left to right. We first find "bc". Since "bc" is in the r-group W, we know that they have always been contiguous. Since they are again contiguous in the demo, we can place them in a TEM-group. Now, we must validate "bc". Since "bc" is currently in an r-group, we know that it has been reversible in the past. Thus, we create an r-group {bc}. We note that the current direction of {bc} is the same as in W. Finally, we can label {bc} with an id, P, for example.

Next, we find the contiguous group "cd". However, when we try and validate "cd", we see that constructing a group with only c and d, would break the {bc} group we just made. Thus, we reject "cd".

There are no more groups of size 2 to find, so we next find the contiguous group "bcd". Since making this group uses all of the previously made P group, we can successfully make a new r-group: {{bc}d} which we can label Q. We note that the current direction of {{bc}d} is the same as in W. We can now see that this situation

requires absorption. Q and P have the same direction relative to W. Moreover, both Q and P have just been created during the processing of W. Thus, Q must absorb P. Now, Q is {bcd}.

Finally, we find the contiguous group "ebcd". Forming this group does not require splitting apart Q. The direction of "eQ" is the opposite of its direction in W:{Qe}. Thus, we should not absorb Q. Instead, we form a new r-group labeled R= {e{bcd}}.

Now, there are no more groups to be found. Thus, we must find all the pieces of W: a and R={e{bcd}}. Since there are two pieces, we re-label them as pieces of W, W^2. The new demo is now "onklmjideaf{e{bcd}}" = "onklmji$X^3X^3W^2X^3W^2$".

Example 1 (after processing <ijkl>):

U-Group: [unordered]
S-Group: <sequential>
R-Group: {reversible}

D_{i+1}= "onklmjifhagbcde"

= "onklmji$X^3X^3W^2X^3W^2$" becomes

D_{i+1}= "on<kl>m{ji}fhag{e{bcd}}"

= "onY^2m $Y^2X^3X^3W^2X^3W^2$"

When processing <ijkl> we find the contiguous group "ji" and create an r-group {ji}. We find the contiguous group "kl" and create an s-group <kl>. We label both of these groups as pieces of Y: Y^2.

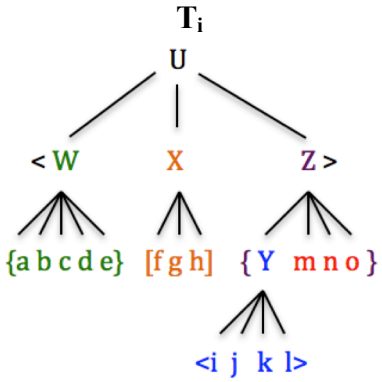
Example 1 (after processing {Ymno}):

D_{i+1}= "onklmjifhagbcde"

= "onY^2m $Y^2X^3X^3W^2X^3W^2$" becomes

D_{i+1}= "{on[<kl>m{ji}]}fhag{e{bcd}}"

= "Z$X^3X^3W^2X^3W^2$"



When processing {Ymno}, we first find the contiguous group "on". We create an r-group S={on} and we mark that this group is in the reverse order as compared to how it appears in current TEM.

Next, we find the much larger contiguous group "Y^2mY^2"="<kl>m{ji}". Since this group contains pieces, we must form an unordered u-group T=[Y^2mY^2]=[<kl>m{ji}]. Notice that neither of the pieces of Y, Y^2, can be grouped with m without also being grouped with the other piece.

Finally, we find all the children of Z contiguously "onY^2mY^2" = "{on}[Y^2mY^2]" = "ST". Note, that in current TEM, this group appears as "Y^2Y^2mno" = "[Y^2mY^2]{on}" = "TS". Thus, we create an r-group V={ST} and mark this direction as opposite. Since both S and V are r-groups created in the processing of Z and since both S and V have been marked with the same relative order, V must absorb its child S. We are left with V = {on[Y^2mY^2]}.

V is the only piece of Z, so we re-label V as Z={on[Y^2mY^2]}.

U-Group: [unordered]
S-Group: <sequential>
R-Group: {reversible}

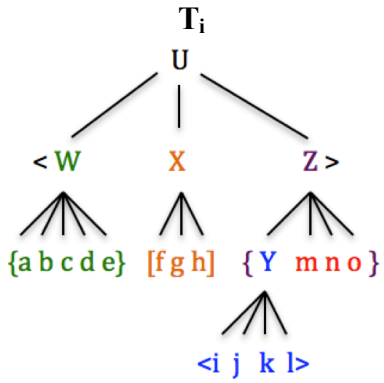
Example 1 (after processing {WXZ}):

D_{i+1}= "onklmjifhagbcde"

= "$ZX^3X^3W^2X^3W^2$" becomes

D_{i+1}= "{ { on[<kl>m{ji}] } [fhag{e{bcd}}] }"

= "U"

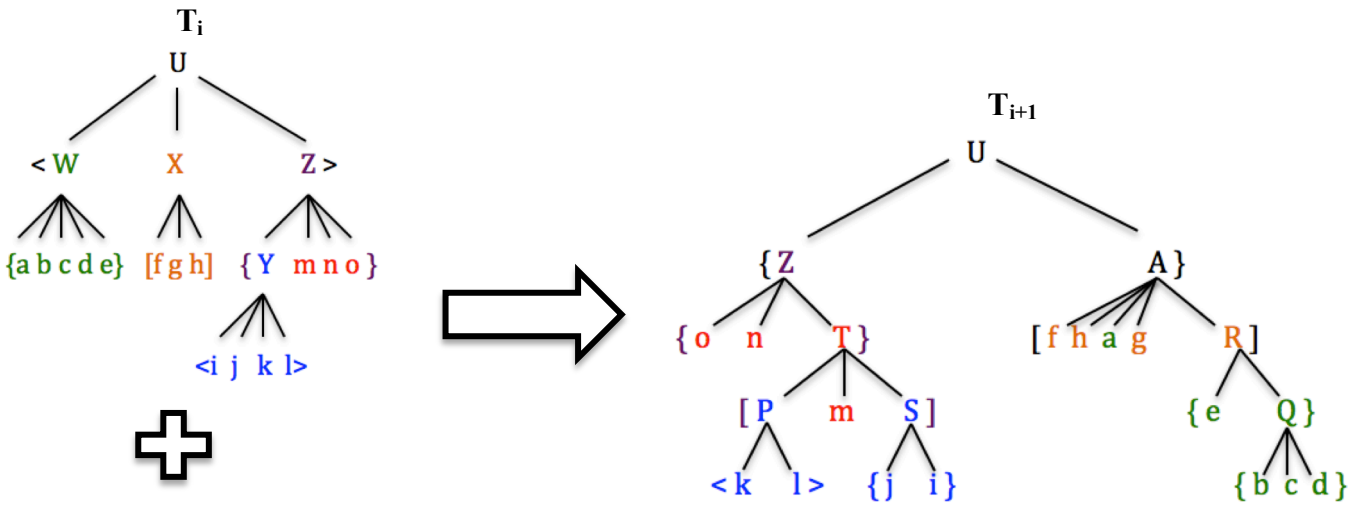


When processing U into "$ZX^3X^3W^2X^3W^2$", we first find the group "$X^3X^3W^2X^3W^2$", which is a u-group A=[$X^3X^3W^2X^3W^2$], as it contains pieces. Finally, we find the entire demonstration as a single contiguous group, U={ZA}.

Example 1 Overview

Figure 4.2

U-Group: [unordered]
S-Group: <sequential>
R-Group: {reversible}



D_{i+1}: "onklmjifhagbcde"

CHAPTER 5

EXPERIMENTAL VALIDATION

We validate the TEM and its construction as follows:

> First, we generate an initial TEM. We can then perform a lazy traversal of the TEM, randomly choosing from the possible actions at each step in the traversal. These random traversals are used to incrementally construct a new TEM. Eventually, the new TEM will be equivalent to the initial one.

We constructed an initial TEM, TESTER, such that there existed:

1. A variety of node structures including:
    a. R-groups: {<><><>}, {}{}{}, {[][][]}, and {[]{}<>}
    b. U-groups: [<><><>], [{}{}{}], [[][][]], and [[]{}<>]
    c. S-groups: <{}{}{}>, <[][][]>, and <{}[]>
2. R-groups, s-groups, and u-groups with their minimum number of children: 2, 2, and 3, respectively.
3. R-groups, s-groups, and u-groups with up to 6 children.
4. All 16 possible types of 3-level nesting, such as:
    a. An r-group containing a child r-group, r, such that r contains an r-group
    b. A u-group containing a child u-group, u, such that u contains a u-group
    c. A u-group containing a child s-group, s, such that s contains an r-group

TESTER had a depth of 8 and contains 125 actions.

TESTER was generated using 3 pre-determined demonstrations.

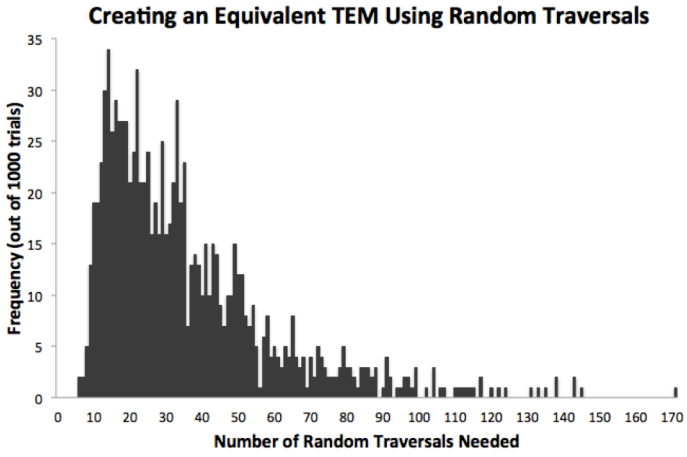
EXPERIMENT:

We repeated the validation experiment 1000 times. In each experiment, we used random traversals from TESTER to create a new TEM, until the new TEM represented the same groups as TESTER. After each experiment, we noted how many random traversals were needed to recreate TESTER. The experiment was performed on a MacBook Pro running OSX 10.8.5 with 2.3GHz Intel Core i5.

RESULT:

On average, it took 35.906 random traversals to recreate TESTER with a standard deviation of 24.402. However, as you can see in Figure 5.1, the data is skewed to the right. The median number of traversals is only 29. At best, only 6 random traversals were needed. However, as previously explained, three carefully chosen demonstrations would suffice to recreate TESTER.

Figure 5.1



During the 1000 experiments, there was a total of 35906 iterations, where an iteration is defined as randomly traversing TESTER, processing the traversal into another TEM, and comparing the two TEMs for equality. The total experimentation time was 900.136 seconds, meaning that each iteration was performed in 0.02507 seconds, on average.

CHAPTER 6

CONCLUSION

TEMs show promise as a model that can automatically detect the natural decomposition of demonstrations into subtasks. However, due to the limitations currently placed on the input demonstrations, TEMs are only appropriate to model tasks that comprise a unique set of actions, all of which must be executed. At best, if a task has several such execution paths, the task can be modeled using several TEMs.

Another limitation of TEMs is that a parallelizable action that can be performed at anytime within a demonstration will be able to break all contiguous groups. In such a situation, the resulting TEM will flatten to a single unordered group. Hence, TEMs are not useful for modeling several parallelizable tasks at the same time.

However, due to their low computational cost, TEMs may still prove useful when utilized to preprocess demonstrations. For example, since actions that pertain to the same task tend to yield a structurally interesting TEM, constructing TEMs from various subsets of actions could help detect parallelizable tasks whose actions have been interwoven.

More significantly, TEMs hint at the benefits of modeling the contiguity or proximity of groups of actions. For example, a more flexible task model could simply keep statistics on the distance between every pair of actions. When a sequence of actions is proposed, every pair of actions could be checked to ensure their proximity matches with some of previous demonstrations.

Modeling proximity or contiguity of actions has the potential to act as a beneficial supplement to existing Learning from Demonstration techniques.

REFERENCES

[1] S. Osentoski, B. Pitzer, C. Crick, G. Jay, S. Dong, D. Grollman*, et al.*, "Remote Robotic Laboratories for Learning from Demonstration," *International Journal of Social Robotics,* vol. 4, pp. 449-461, 2012.

[2] E. Guimarães, A. Maffeis, J. Pereira, B. Russo, E. Cardozo, M. Bergerman*, et al.*, "REAL: A virtual laboratory for mobile robot experiments," *Education, IEEE Transactions on,* vol. 46, pp. 37-42, 2003.

[3] B. D. Argall, S. Chernova, M. Veloso, and B. Browning, "A survey of robot learning from demonstration," *Robotics and Autonomous Systems,* vol. 57, pp. 469-483, 2009.

[4] P. E. Rybski, K. Yoon, J. Stolarz, and M. M. Veloso, "Interactive robot task training through dialog and demonstration," in *Human-Robot Interaction (HRI), 2007 2nd ACM/IEEE International Conference on*, 2007, pp. 49-56.

[5] G. Andrew, R. Kathy, and R. Charles, "Learning hierarchical task models by defining and refining examples," presented at the Proceedings of the 1st international conference on Knowledge capture, Victoria, British Columbia, Canada, 2001.

[6] H. Veeraraghavan and M. Veloso, "Teaching sequential tasks with repetition through demonstration," in *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 3*, 2008, pp. 1357-1360.

[7] N. N. Monica and J. M. Maja, "Natural methods for robot task learning: instructive demonstrations, generalization and practice," presented at the Proceedings of the second international joint conference on Autonomous agents and multiagent systems, Melbourne, Australia, 2003.

[8] M. N. Nicolescu and M. J. Matarić, "A hierarchical architecture for behavior-based robots," in *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 1*, 2002, pp. 227-233.

[9] N. Nejati, P. Langley, and T. Konik, "Learning hierarchical task networks by observation," in *Proceedings of the 23rd international conference on Machine learning*, 2006, pp. 665-672.

[10] C. Hogg, H. Munoz-Avila, and U. Kuter, "HTN-MAKER: Learning HTNs with Minimal Additional Knowledge Engineering Required," in *Aaai*, 2008, pp. 950-956.

[11] T. Russell, S. Halit Bener, and C. Sonia, "A practical comparison of three robot learning from demonstration algorithms," presented at the Proceedings of the seventh annual ACM/IEEE international conference on Human-Robot Interaction, Boston, Massachusetts, USA, 2012.