# Task Recognition and Generalization in Long-Term Robot Teaching

Guglielmo Gemignani[1⋆], Steven D. Klee[2⋆], Daniele Nardi[1], and Manuela Veloso[2]

[1]Department of Computer, Control, and Management Engineering "Antonio Ruberti", Sapienza University of Rome, Rome, Italy
{gemignani,nardi}@dis.uniroma1.it
[2]Computer Science Department, Carnegie Mellon University 5000 Forbes Ave., Pittsburgh, PA 15213, United States {sdklee,veloso}@cmu.edu

**Abstract.** Several research efforts have addressed the challenge of having non-technical users teach or demonstrate a task to a robot, focusing on the teaching method and the task acquisition. In this paper, we consider an autonomous robot that persists over time interacting with users and the problem of teaching an *additional* task to the robot. We believe that the assumption that a user would know all the tasks previously taught to the robot does not hold. We hence investigate the problem of recognizing when a user is teaching a task similar to one it already knows. We present a graph-based task representation, and contribute algorithms to measure task similarity, to perform task generalization, and to generate task proposals, as a user teaches new task actions to an agent. Tasks are accumulated in a library, and the agent can learn and represent common patterns among tasks. We experimentally demonstrate our methods in a robot manipulator and a mobile service robot with large task libraries. We show that our methods lead to a significant reduction in necessary user-agent interaction and average task representation size.

**Keywords:** Instruction Graphs; Task Autocompletion; Robot Planning and Plan Execution; Robotic Agent Languages and Middleware for Robot Systems

## 1 Introduction

Several research efforts have focused on creating robots that learn from non-technical users through natural language interaction. In this paper, we address the problem of a user teaching an *additional* task to a service robot that is deployed for an extended period of time. In such cases, the user is unlikely to know all of the tasks previously taught to the robot. Therefore, we want the robot to recognize when it is being taught a task similar to one that it already knows. In these cases, the robot will use this information to propose the next steps of a task to the user, reducing the necessary number of human-robot interactions.

⋆ The first two authors have contributed equally to this paper

For a long-term deployed robot, we assume that its task library will become very large. So, it is not feasible to compare a newly taught task to every past entry. Since many tasks have similar patterns of action and sensing primitives, instantiated with different parameters, we instead focus on learning a library of general parametric tasks. Specifically, the robot learns classes of tasks from the bottom-up with minimal user assistance. We show that by creating a general task library we can perform online generalization and task step proposals for libraries with tens of thousands of features. Moreover, these parametric tasks can be referenced at teaching-time like programming functions, instead of creating a specific instance of the task from scratch. This greatly reduces the overall size of each task being taught if parts of it belong to a general class.

In this paper, we contribute an approach for measuring task similarity, performing task generalization, and proposing future steps of a task during the teaching process. We represent tasks as graph-based structures called Instruction Graphs [4]. We perform frequent labeled subtree mining [11] to extract general classes from a task library and use a structure-based similarity metric to propose the most likely class.

The main contributions of this paper are:

- An algorithm to generalize similar tasks.
- A structure-based approach for recognizing task similarity in task teaching frameworks.
- A method for robots to propose the next steps of a task during teaching.

In the rest of the paper we first give a brief overview of related work, underlining the main differences with our approach. We then describe our Generalization and Proposal approach, showing all of our contributions thoroughly. Then, we discuss the evaluation process undertaken to validate the approach. Lastly, we conclude the paper with a discussion of our contributions and remarks on possible future work.

## 2 Related Work

The problem of teaching a robot new tasks by interacting with a user is a research topic that has been drawing increasing attention in the last few years. One of the first approaches that tackles such a problem focuses on creating complex-tasks through the combination of pre-existing robot action primitives [2]. This work however, allows the user to specify a task only as a sequence of primitives.

More recent works address this problem, by focussing on converting user utterances into more complex task representations. For example, one approach represents tasks as acyclic graphs, which are composed of nodes representing finite state machines [3].

Recently, a more expressive teaching framework based on Instruction Graphs has been presented [4] to support the teaching of tasks involving loops and conditionals. This task teaching approach also allows the user to apply corrections

to the previously learned tasks. However, like the previous approaches, it only handles instantiated tasks.

The majority of the works published do not generalize over the tasks learned, only aiming at teaching to a robot multiple instantiated tasks. An alternative approach has been proposed to handle parametric tasks [5], where the parameters are represented as variables that are specified at runtime. This differs from our approach because the users describe the parametric tasks, instead of learning from examples.

A previous work proposes to divide the learning approach in two main phases [6]. In a first phase, the user is able to give a demonstration of the action to be learned. In a second phase, the robot has the opportunity to refine the acquired capabilities by practicing trials under the teacher's supervision. In this work, the generalization process starts with a set of examples of the same task. By contrast, we start with a library of many examples of different tasks. We also attempt to generalize similar pieces of different tasks.

Many of the task teaching work in literature rely on a graph-based task representations. However, there are also examples of approaches representing tasks as formal logic goal descriptions [7]. In this work the instructions received from the user are associated with formal logic expressions that represent the task taught to the robot. In this work, we only consider structure based generalization for graph-based task representations.

For graph representations, generalization from examples requires finding common subgraphs between different tasks. The general problem of finding labeled subgraph isomorphisms is NP-Hard [8]. However, such task becomes tractable if, instead of graphs, the same problem is faced when dealing with trees. In fact, multiple algorithm have been proposed for mining common frequent subtrees from a forest (i.e., a library) of trees.

One of the most referenced frequent subtree miner is Treeminer [1]. This algorithm uses equivalence class based extensions to quickly discover frequent embedded subtrees.

Another algorithm that is able to mine for frequent paths, subtrees and subgraphs is GASTON [9]. The algorithm provides a solution by splitting up the frequent subgraph mining process into path mining, then subtree mining, and finally subgraph mining.

For the purpose of this paper we adopted instead SLEUTH, an open-source frequent subtree miner, able to efficiently mine frequent, unordered or ordered, embedded or induced subtrees in a library of labeled trees [10]. This algorithm uses scope-lists to compute the support of the subtrees while adopting a class-based extension mechanism for candidate generation. For a more complete description of the available frequent subtree mining algorithm we refer the reader to [11].

In the following section we show a way of generalizing over Instruction Graphs by showing how, starting from a library of taught tasks, we are able to identify common frequent subtasks that are later generalized with the aid of the user.

## 3 Approach

In this section, we first give a brief overview of Instruction Graphs, a graph-based representation used for representing tasks taught by a user to a robot through natural language interaction. Subsequently, we briefly describe the speech interface adopted and present an in depth description of the generalization and task proposal algorithm. For the remainder of the work we use the terms agent and robot interchangeable, since our approach can also be applied to virtual agents.

### 3.1 Instruction Graphs

In this work, we generalize over Instruction Graphs. In the Instruction Graph representation, each task is identified with a unique integer value called `GraphID` and is represented as a collection of robot sensing and action primitives, ordered in a directed graph structure. Mathematically, an Instruction Graph is a tuple $G = \langle V, E \rangle$ where $V$ is a set of vertices and $E$ is a set of edges connecting the vertices. In this graph, vertices represent actions to be executed by the robot, and edges represent possible transitions to the next actions. Each vertex is represented as a 3-tuple denoted as $v = \langle$NodeID, InstructionType, Action$\rangle$.

The $NodeID$ is a unique integer value, used to reference a specific node in the graph. By convention, the vertex with $NodeID = 0$ is the starting vertex of the graph from which the robot begins execution while the $NodeID = -1$ is the vertex that denotes the ending of a task.

The *Action* of a vertex is defined by a 2-tuple $Action = \langle f, p \rangle$, consisting of a function $f$ called the `execution function` and a list of parameters $p$. The `execution function` represents the action the robot will perform in this state parameterized by a list of parameters. Each parameter is, in fact, an instance of a class representing the semantic meaning of the variable. For example, a floating point number could belong to the class `Angle` or the class `Distance`. These classes are domain specific, and are used during the dialog with the user, and for task generalization.

The $InstructionType$ of the vertex describes how the robot should interpret the output of a its *Action*. This interpretation affects how the robot transitions through the instruction graph during execution. The Instruction Graph representation has the following $InstructionTypes$:

- **Do**: Used for a vertex where the *Action* field refers to open-loop actions. These actions are performed with no sensory information.
- **DoUntil**: Used for a vertex where the *Action* field refers to a closed-loop action. These actions are performed until a sensory condition is true.
- **No-Op**: Used for scoping conditionals and loops. Performs no action and immediately transitions to the next node.
- **Conditional**: Used for a vertex where the *Action* field can be evaluated to a boolean value. These create branches in execution. Both branches eventually reconnect to a `No-Op` node that signals the end of the conditional structure.

– **Loop**: Used for a vertex where the *Action* field can be evaluated to a boolean value. Vertices inside the loop structure will be repeatedly executed while this condition is true. The end of this structure is delimited by a `No-Op` node with an edge pointing back to the beginning.
– **Reference**: Used to include another Instruction Graph in the current one. The `execution function` is instantiated with a function that returns a reference to the linked Instruction Graph.

To better understand these nodes, Figure 1 shows an example of each *InstructionType*, besides the `Reference`.

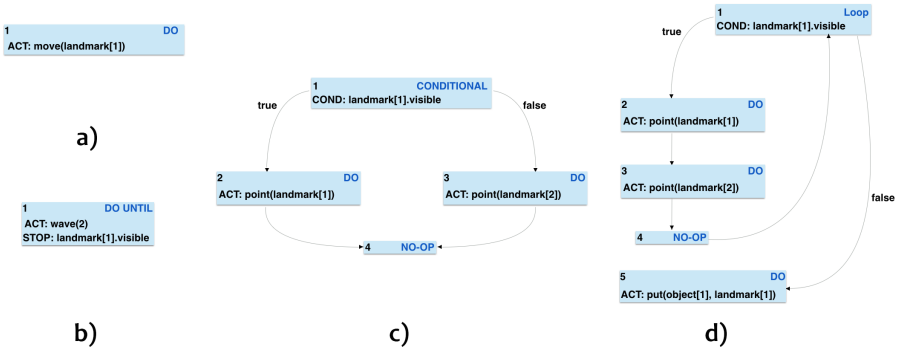


Fig. 1: Examples of InstructionTypes and structures defined in our Instruction Graph representation: a) Do; b) DoUntil; c) Conditional structure; d) Loops structure.

The Instruction Graphs are constructed through a natural language interface with the user. We use a probabilistic parser and grounded to process natural language [12]. Starting from a base grounding, the robot learns the groundings from natural language to environmental features, robot primitives, and parts of the task representation. The robot improves these groundings by asking the user when it is unsure of how to ground an expression. Once learned, an Instruction Graph can be saved and used as a sub-routine in a larger task, through the `Reference` node.

The Instruction Graphs generated through the interaction with the user are executed as a graph traversal operation starting from the node with ID 0 (i.e., the start or root node). At each step, the robot executes the *Action* of the current vertex and transitions based upon the result and the vertex's *InstructionType*. The robot transitions according to the following rules:

– **Do, DoUntil**: the *Action*'s return value is ignored and the robot transitions to the next out-neighbor. These actions will only have one neighbor.
– **Conditional**: the *Action*'s return value is a boolean that determines whether the robot should transition to the first or the second neighbor.

- **Loop**: the *Action*'s return value is a boolean that determines whether the robot should transition to the first or second neighbor. The true case transitions to the first neighbor, which will execute code in a loop that eventually returns to this vertex. The false case transitions to the second neighbor which will exit the loop and proceed with the execution of the remaining part of the Instruction Graph.
- **Reference**: The *Action* field is used to transition to the start node of the referenced Instruction Graph. When such a referenced graph reaches the stop node, the agent transitions to the neighbor of the Reference node.

### 3.2 Generalizing Instruction Graphs

In this section we describe an algorithm for extracting general classes of tasks from a library of Instruction Graphs. Then we discuss how these general classes can be used by an agent to propose future actions during task teaching. These general task classes are represented as parametric tasks where their arguments are expressed at run-time. For example, let's consider the task of picking up an object and placing it at a particular location. This task has two implicit parameters: the object to be moved and its target final position. In this section, we show how an agent can learn a parametric task from a library containing multiple instantiated examples of tasks. More formally, we define a parametric task as an Abstract Instruction Graph, which is an Instruction Graph where the parameters of a node's *Action* are not all instantiated. These abstract, or open, parameters have a type such as *Joint*, *Angle*, or *Distance*, but no value.

Our approach is composed of three steps. First, we find clusters of similar tasks and extract common patterns. Then, we learn which action parameters are part of the pattern and which should be parameterized, creating general classes of tasks. Lastly, in future situations, we allow the agent to recognize when a task being taught contains one of the general classes, and propose it to the user.

**Extracting Subgraph Patterns** There are many ways to recognize the common parts of multiple tasks, but most of them are domain specific. For instance, on a manipulator we might consider how similar the arm motions are. However, this measurement is not informative on a mobile base with no arms. To find similarities between tasks in a general manner, we look for patterns in the task representation. Since tasks are represented as graph-like structures, this problem is closely related to the general problem of enumerating labeled subgraph isomorphisms, which is NP-Hard. However, for rooted labeled trees where each vertex has exactly one parent, this problem becomes tractable. For this reason, when looking for task generalizations, we relax the problem by computing a unique spanning tree for each task.

For Instruction Graphs, we define an injective function to create these spanning trees by recursively modifying all the conditional and loop structures in the graph. In particular, to ensure that each node has exactly one parent, we perform a depth first search through the graph, removing back edges. For nodes with two

children such as conditionals and loops, the depth first search is deterministic such that two structural similar IGs have the same spanning tree.

To find patterns in these tasks, we use this forest of trees as input to a frequent labeled subtree mining algorithm. These algorithms return frequent labeled subtrees and mappings from each node in the pattern to the nodes in the input forest. A frequent labeled subtree can be defined as a labeled tree that has an occurrence frequency greater than a certain threshold in a given forest. Such a frequency is usually called the `support` of the tree. In our case, the label of a node is a tuple $(I, F)$ where $I$ is the `ActionType` of the vertex and $F$ is the `Execution Function`. With this formulation, a frequent subtree corresponds to a collection of nodes that appear in multiple Instruction Graphs, possibly with different parameters. The support, instead, represents the minimum number of examples needed for a certain task to be considered for generalization. Ideally, the support should be set to the frequency of the smallest class in the input forest. By lowering the parameter, our computational cost increases because we generate many classes that are too specific. By increasing the support, the computational cost decreases, but we may miss some classes.

Once the tree patterns are obtained, they are then filtered to eliminate any that are redundant or unwanted. In particular, we discard tree patterns with incomplete conditional or looping structures since they cannot be executed by a robot. A structure is defined incomplete if one of its `Conditional` or `Loop` nodes is not connected to its corresponding scoping `No-Op` node. Figure 2 shows an example of incomplete structure obtained from the generalization of two different instruction graphs. The remaining subtrees are again filtered by discarding any pattern that is completely contained in another one. With this filter we prefer to generalize classes with the largest number of nodes. Note that for every pattern, there are many sub-patterns containing fewer nodes. This filter is necessary to greatly reduce the computational complexity of this approach. However, it can lead to ignoring classes of tasks that are completely contained in another larger class. When the final list of frequent common subtrees is obtained, it is processed to create general task classes.

**Building General Classes from Tree Patterns** Given a collection of example Instruction Graphs whose spanning trees contain a particular pattern, our next step is to create a general class. In particular, this involves creating a parameterized task (i.e., Abstract Instruction Graph) and using the specific examples to learn which instantiated parameters are part of the class, and which are open parameters.

First, the Abstract Instruction Graph is generated by extracting the subgraph related to the particular pattern. To do this we perform a depth first search through the subtree pattern and copy the structure of any one of the specific Instruction Graphs the pattern matched. Once this is complete, we have the structure of the general class, but still need to determine which parameters are open.

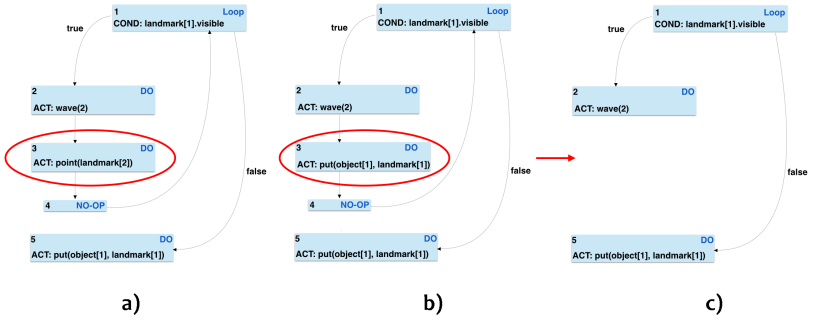


Fig. 2: Example of two Instruction Graphs **a** and **b**, that are generalized to an incomplete structure. The nodes circled in red have different actions and are not part of the generalized task. The resulting Instruction Graph **c** can not be run by a robot.

To do this, we define an $\epsilon$-threshold. For each parameter present in each node of the Abstract Instruction Graph, we check the corresponding value in all of the examples. If the parameters have the same value at least $\epsilon$ percent of the time, it becomes part of the Abstract Graph as an instantiated parameter. Otherwise, it is left uninstantiated, to be defined by the user at the task-teaching time. We do this for every subtree pattern found that has not been filtered by our heuristics, creating a general task class library. Algorithm 1 details each step of this procedure for one pattern.

**Algorithm 1** Abstract Instruction Graph Creation

**procedure** CREATEABSTRACTIGS($pattern, IGs$)
  $AbstractIG \leftarrow copy\_dfs(pattern, IGs[random()])$
  **for** $n\_id \in AbstractIG$ **do**
    **for** $p\_id \in len(node.parameters)$ **do**
      **if** $\exists v$ : $\epsilon$ percent of IGs[n_id][p_id] == v **then**
        $AbstractIG[n\_id][p\_id] \leftarrow v$
      **else**
        $AbstractIG[n\_id][p\_id] \leftarrow abstract$
      **end if**
    **end for**
  **end for**
  **return** $AbstractIG$
**end procedure**

Next, the agent should prompt a user to see if these are valid classes. However, we may have many Abstract Instruction Graphs to present to the users. Rather

than overwhelming them, we wait until the next time a person tries to teach it a task that contains part of these classes. At this point, we provide a task proposal, and for new classes, we also ask to confirm that they are valid generalizations.

**Proposing Future Task Steps** During task teaching, the agent learns a task by building an Instruction Graph through natural language interaction with the user. After each step of this interaction, the agent compares the partial task to all of its Abstract Instruction Graphs. Such a comparison is carried out by checking if a predefined initial percentage of the general pattern matches the endings of the current partial task. If so, the agent proposes the next steps of the task to the user. In this way, we are able to recognize and propose general tasks

When a specific proposal is found, the robot notifies the user by displaying a representation of the Abstract Instruction Graph and by asking the user for permission to demonstrate the task. When granted permission, the agent demonstrates the task, asking the user to instantiate the abstract parameters encountered as they become necessary. If multiple proposals are valid, they are ranked by first percentage of similarity with the current task and then by the number of instantiated Instruction Graphs in the class. At this point, we break ties randomly.

At the end of the demonstration, the agent asks the user if this proposal should be merged to the task being taught. If so, the proposal is merged by removing all the nodes present in the pattern and adding a `Reference` node that links to the Abstract Instruction Graph. The `Reference` node has a list of parameters based on those suggested by the user. Before executing the Instruction Graph, we instantiate the open parameters from this list. With this substitution, the length of the task is greatly reduced. If this class has not previously been proposed, the robot also asks for a class name.

Figure 3 shows a sample interaction with the user. In such an interaction, the user starts teaching the robot a task. During the first two interactions, the robot is not able to come up with any suggestion. In this case in fact, no previous task starting with *open_gripper* and *point_at_landmark* is known by the robot. When the third action is instead uttered by the user, the robot recognizes that at least one abstract class starting with *display_message* has been previously extracted from the task library. However, the pattern recognized does not surpass the predefined $\epsilon$-threshold yet. When the fourth instruction is given to the agent, a suggestion that meets the required criteria is finally found and the suggestion procedure is then started. Figure 4 depicts the graph shown to the user when a suggestion for a task being taught is found. At any point, the user can quit a suggestion and resume the task teaching process.

## 4 Experiments

In this section we first describe the actual implementation on a manipulator and mobile base. Then, we experimentally demonstrate how our approach can

```
Exact Textual Interaction

U: Open Gripper
R: I will open my gripper.
R: What should I do next?
U: Point at landmark 1
R: What should I do after pointing
   at landmark 1?
U: Display message "Hello".
R: Ok, what should I do next?
U: Set your left arm to 80 degrees.
R: I think you are teaching me
   something similar to: unnamed class.
R: Can I demonstrate it to you?
U: Yes.
R: First I will display the message "Hello".
R: Then I will set my left arm to 80 degrees.
R: Now I will set my right arm to [x] degrees.
   Can you please provide a value?
U: Set your right arm to 90 degrees.
R: *Performs Action*
R: This is my full suggestion.
R: Would you like to use it?
U: Yes.
R: I also do not know the class name.
R: Can you provide one?
U: Semaphore and Greet
R: Class saved as "Semaphore and Greet".
R: Thank you. Please continue teaching.
```

Fig. 3: Sample conversation between the user during task teaching. In this case, the manipulator finds a candidate proposal after the third action. After the fourth action taught, the $\epsilon$-threshold is met, so the manipulator proposes the task.

extract general tasks from a large task library very quickly. Next, we show that this leads to a significant reduction in the number of human-robot interactions and average task representation size.

### 4.1 Setup

The system has been deployed on a Baxter manipulator robot and on a CoBot that have interacted with multiple user over an extended period of time. The CoBot has an omnidirectional mobile base equipped with sensors including a touch screen, microphones, a laser range finder, camera, and Microsoft Kinect [13]. The Baxter robot has cameras on both arms and is also equipped with a Microsoft Kinect. On CoBot, human-user interaction occurs verbally, and on the manipulator it occurs over a keyboard and monitor. In this experiment, we make

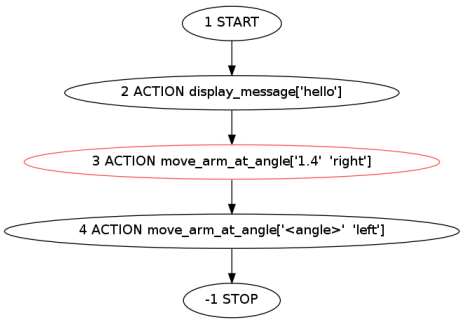


Fig. 4: Sample Instruction Graph displayed to the user during a suggestion for a "Semaphore and Greet" action. The red node represents the action currently being demonstrated. The floating point numbers represent instantiated angle parameters, while the variable `<angle>` represents an abstract parameter.

use of 12 different sensing and action primitives designed for these robots. The complete list of primitives used in this experiment is reported in Table 1. Both robots are shown together in Figure 5. The frequent subtree mining algorithm adopted in the experiment is an open source version of SLEUTH [1].

Table 1: Sensing and Action Primitives Used in the Experiments

| **Primitive Name** | **Robot** | **Sensing or Action** |
|---|---|---|
| *open_gripper* | Baxter | Action |
| *close_gripper* | Baxter | Action |
| *point_at* | Baxter | Action |
| *move_gripper_to_landmark* | Baxter | Action |
| *move_arm_at_angle* | Baxter | Action |
| *display_message* | Baxter | Action |
| *move_forward* | Mobile Base | Action |
| *turn* | Mobile Base | Action |
| *go_to_landmark* | Mobile Base | Action |
| *follow_person* | Mobile Base | Action |
| *say_message* | Mobile Base | Action |
| *is_landmark_visible* | Baxter and Mobile Base | Sensing |

### 4.2 Task Generalization Speed and Accuracy

In order to experimentally evaluate the generalization approach, we first tested the algorithm on a large library of tasks by measuring the number of general

[1] `www.cs.rpi.edu/~zaki/software/`

Fig. 5: Baxter and mobile service robots used during the experiments.

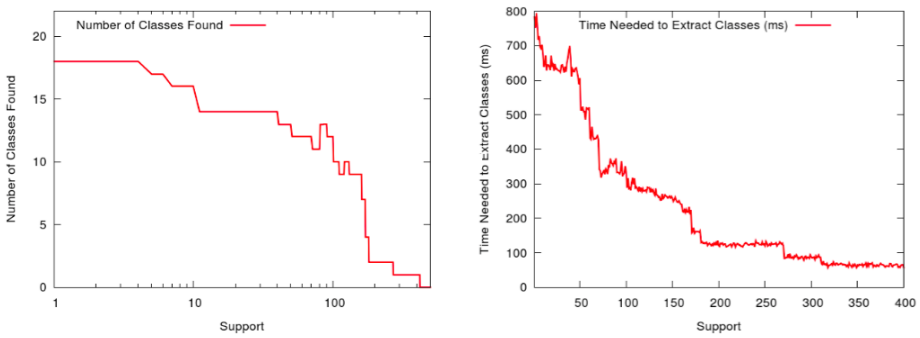


Fig. 6: (a) Plot of Classes Found vs Support and (b) Plot of Time Needed to Extract Patterns ($ms$) vs Support.

task extracted with respect to the support chosen. Such a library was composed by 1000 different tasks belonging to 10 unique classes specifically devised for a manipulator robot (i.e., the Baxter robot in our case) and an additional 10 classes devised for our mobile service robot. The number of instances for each class varied from 10 to 100 different tasks and they were generated both through user interaction and through an automated algorithm. Specifically users taught 1 to 10 tasks for each class, and the others were automatically created by taking the same tasks structures and randomly generating different parameters. Users were given a high level description of each class and told all of the primitives available. For instance, some of the classes for the manipulator were waving, greeting, picking up objects, and performing semaphore messaging. Some of the classes for the mobile base were greeting, going to a location, and following a moving landmark. Figure 6 shows the results obtained from this first experiment.

With the support that generates the most classes (i.e., the optimal support), we were able to generalize 18 of the 20 possible classes, and one additional

unexpected class. It can be noticed that the support that extracts the maximum number of classes is 4 or less. This value is in contrast with the expected value of 10, which corresponds to the number of tasks in the smallest generalized class. This is because a structure-based approach differentiates between sequences of actions that can be composed in an arbitrary order. Such a peculiarity was presented by one of the classes acquired through the interaction with the users. The `Semaphore and Greet` class, in fact, required the Baxter to position its arms at a certain angle in order to perform one of the flag semaphore signs as depicted in Figure 7. The robot would also display the message "Hello" on its screen. This task is composed by a sequence of two *move_arm_at_angle* primitive actions needed to move the two arms, and a *display_message* action. Such a sequence of actions was taught by different users in two different ways: some users told the robot to display the message and then move its arms, and the others displayed the message after moving both of its arms. This class was therefore generalized as two different general tasks instead of one.

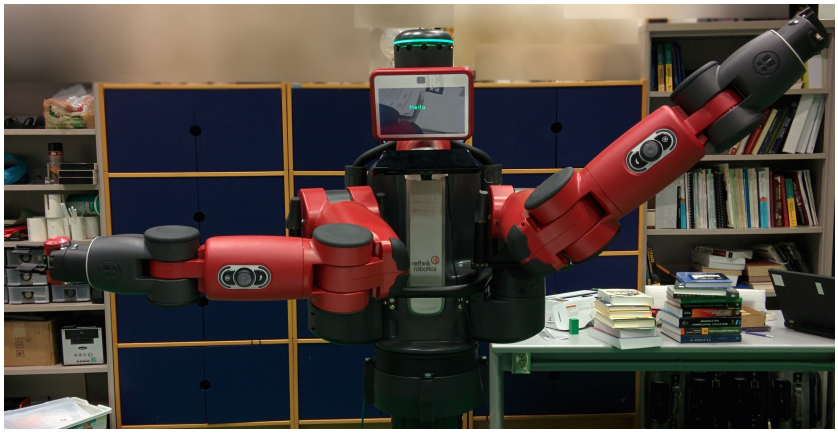

Fig. 7: Baxter robot performing the flag semaphore sign representing the letter Q and displaying the message "Hello".

The three classes we were unable to generalize at the optimal support value were missed because they are subclasses of another class. As previously described, since we filter any subgraphs that are completely contained in another one, we only generalize the tasks that have the maximum possible number of nodes. Such a greedy filtering was performed in order to avoid an exponential explosion in the number of patterns found by the algorithm. However, it can also be noticed that our approach still finds some of these classes after reaching a support greater than 100, the maximum size of any class. This occurs because these patterns are shared between multiple classes, and therefore have a much larger support. For example, our Waving class on Baxter is a subsumed by the Greeting class that waves and says a message. Normally, the Waving class is not

generalized, because it is ignored by our filters in favor of the Greeting class. However, when the support is over 100, we no longer generalize the Greeting class, so the Waving class is not filtered away. Similarly, a pattern found in two classes may only become visible when the support is greater than the size of the larger class. This explains why our graph of generalized classes versus the support is not monotonically decreasing. It also gives us intuition that a larger support leads to finding more general classes.

As a second experiment we analyzed the speed of the generalization process while varying the value of the support. Figure 6b shows the results obtained. In particular, using larger supports leads to a much faster computation due to the fewer possible patterns found to be generalized. Even for a support of one, a task library with one thousands tasks can be generalized in under a second.

For a third experiment, we decided to test the effect of varying the number of entries in the graph library, while holding the support and number of classes constant. We wanted to ensure that our algorithm could work on extremely large task libraries. Using the same classes as above, we increased the number of graphs in each class. For a graph library with $10,000$ tasks, we could generalize the same number of classes in 4 seconds for a support of 1. For a graph library with $100,000$ tasks, we could generalize the same number of classes in 37 seconds for a support of 1.

### 4.3 Impact on Human-Robot Interactions and Representation Size

Next, we want to consider the impact of this approach on task representation size, and the amount of necessary interactions. Referencing general classes, instead of constructing instantiated version from scratch, can only decreases the size of a task. For our test library, the average size of the general classes was 7 nodes excluding the start and stop nodes. Assuming an action always belongs to a class, and that we choose the class randomly, we expect to save 6 nodes using task proposal.

Now, we focus our analysis on the number of user interactions required with, and without task proposals. We define a user-interaction to be one series of questions and answers between the agent and the user. For task teaching without proposals, we require at least one interaction for every component of the task. More interactions are necessary if the user makes a mistake and has to correct a previous command. With task proposals, we have one interaction for the robot to provide the initial suggestion, and an additional interaction for each open parameter. Clearly, this reduces the need for human input when the agent's proposals are correct.

To ensure that an agent's task proposals are usually correct, we can adjust the $\epsilon$-threshold that determines what percentage of the graph must match the pattern before it is suggested. If this parameter is too low, the suggestions will likely be poor quality. If it is too high, the suggestions will occur too late in the teaching process to significantly reduce the number of interactions. In practice, we have found that the parameter is affected by how similar different generalized classes are. If some classes are very similar, it typically needs to be much higher.

In the future, we have considered having different suggestion thresholds for each generalized class, based on its similarity to other generalized classes.

## 5 Conclusion and Future Work

In this paper, we presented an approach for generalizing collections of tasks taught to a robot over an extended period of time. This bottom-up technique allows an agent to learn parameterized tasks from collections of specific examples, based on task structure. The main contributions are a method for finding generalized tasks using frequent labeled subtree mining algorithms, and an approach for agents to propose future steps of a task to reduce human-robot interactions and the size of each task. By mapping our tasks to trees, we are able to handle the difficulty normally associated with finding subgraph isomorphisms.

Our approach is implemented modularly, so that any labeled tree miner can be used to find patterns. Furthermore, we have implemented the technique on a mobile base and manipulator robot, but it can work on any agent running the Instruction Graph task teaching framework.

For the experiments performed, we showed that our technique effectively generalizes most of the classes in a large test library. Additionally, by measuring the computational time, we showed that our approach is fast enough to be run online for task libraries on the order of thousands of tasks in size. When run offline, this approach scales to libraries containing hundreds of thousands of graphs.

As a future work, we plan to extend the framework to support domain-specific generalization. For instance, we might use the motion patterns of a manipulator as a measure of task similarity. Additionally, we are working to allow users to actively query the robot for the tasks it knows. These queries may be goal-based or involve domain-specific information associated to each task.

## References

1. Zaki, Mohammed J: Efficiently Mining Frequent Trees in a Forest. Proceedings of the 8th ACM International Conference on Knowledge Discovery and Data Mining. (2002)
2. Lauria, S., Bugmann, G., Kyriacou, T., Bos, J., Klein, E.: Personal Robot Training via Natural-Language Instructions. IEEE Intelligent Systems. (2001)
3. Rybski, P.E., Yoon, K., Stolarz, J., Veloso, M.M.: Interactive Robot Task Training Through Dialog and Demonstration. Proceedings of HRI. (2007)
4. Meriçli, Ç., Klee, S.D., Paparian, J., Veloso, M.: An Interactive Approach for Situated Task Specification Through Verbal Instructions. Proceedings of the 13th International Joint Conference on Autonomous Agents and Multiagent Systems. (2014)
5. Connell, J., Marcheret, E., Pankanti, S., Kudoh, M., Nishiyama, R.: An Extensible Language Interfacefor Robot Manipulation. Artificial General Intelligence. (2012)

6. Nicolescu, M.N., Matarić, M.J.: Natural Methods for Robot Task Learning: Instructive Demonstrations, Generalization and Practice. Proceedings of the 2nd International Joint Conference on Autonomous Agents and Multiagent Systems. (2003)
7. Dzifcak, J., Scheutz, M., Baral, C., Schermerhorn, P.: What to Do and How to Do It: Translating Natural Language Directives into Temporal and Dynamic Logic Representation for Goal Management and Action Execution. Proceedings of ICRA .(2009)
8. Kimelfeld, B. and Kolaitis, P.G.: The Complexity of Mining Maximal Frequent Subgraphs. Proceedings of the 32nd Symposium on Principles of Database Systems. (2013)
9. Nijssen, S., Kok, J.N.: A Quickstart in Frequent Structure Mining Can Make a Difference. Proceedings of the 10th International Conference on Knowledge Discovery and Data Mining. (2004)
10. Zaki, M.J.: Efficiently Mining Frequent Trees in a Forest: Algorithms and Applications. IEEE Transactions on Knowledge and Data Engineering. (2005)
11. Jiang, C., Coenen, F., and Zito, M.: A survey of Frequent Subgraph Mining Algorithms. The Knowledge Engineering Review. (2013)
12. Kollar, T., Perera, V., Nardi, D., and Veloso, M.: Learning Environmental Knowledge from Task-based Human-robot Dialog. ICRA. (2013)
13. Biswas J. and Veloso, M.: "Localization and Navigation of the CoBots Over Long-Term Deployments", in The International Journal of Robotics Research. (2013)