Graph-Based Task Libraries for Robots: Generalization and Autocompletion

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. In this paper, we consider an autonomous robot that persists over time performing tasks and the problem of providing one *additional* task to the robot's task library. We present an approach to *generalize tasks*, represented as parameterized graphs with sequences, conditionals, and looping constructs of sensing and actuation primitives. Our approach performs graph-structure task generalization, while maintaining task executability and parameter value distributions. We present an algorithm that, given the initial steps of a new task, proposes an autocompletion based on a recognized past similar task. Our generalization and autocompletion contributions are effective on different real robots. We show concrete examples of the robot primitives and task graphs, as well as results, with Baxter. In experiments with multiple tasks, we show a significant reduction in the number of new task steps to be provided.

1 Introduction

Different mechanisms enable robots to perform *tasks*, including directly programming the task steps; providing actions, state, goals or rewards, and a planning algorithm to generate tasks or policies; and instructing the tasks themselves in some representation. In this work, we consider tasks that are explicitly provided and represented in a graph-based task representation in terms of the robot's sensing and actuation capabilities as *parameterized primitives*.

We address the problem of efficiently giving an *additional* task to a robot that has a task library of previously acquired tasks. We note that just by looking at a robot, one does not know the tasks the robot has already acquired and can perform. The new task may be a repetition, or a different instantiation, or be composed of parts of other known tasks. Our goal is to enable the robot to recognize when the new task is similar to a past task in its library, given some initial steps of the new task. Furthermore, the robot recommends *autocompletions* of the remaining parts of the recognized task. The robot recommendations include

* The first two authors have contributed equally to this paper

proposed parameter instantiations based on the distribution of the previously seen instantiated tasks.

The contributions of the work consist of the introduction of the graph-based *generalized task* representation; the approach to determine task similarity in order to generate a library of generalized tasks; and the algorithm for recognition of the initial steps of an incrementally provided task, and for proposing a task completion. The task and the generalized tasks are represented as a graph-based structure of robot action primitives, conditionals, and loops.

For generalizing graph-based tasks, and since the general problem of finding labeled subgraph isomorphisms is NP-hard, our approach includes performing subtree mining in a unique spanning tree representation for each graph. The tree patterns, which are executable by the robot and frequent, are added to the library. The saved patterns are generalized over their parameters, while keeping the distribution of values of the parameters of the corresponding instantiated task graphs. The library is then composed of the repeated parts of the given graph-based generalized tasks.

When a new task is given to the robot, the robot uses the library of generalized tasks to propose the next steps with parameters sampled from its distribution, performing *task autocompletion*. The robot performs the proposed completions, which are accepted as a match to the new task or rejected. The process resembles the autocompletion provided by search engines, upon entering initial items of a search. The task autocompletion in our robots can incrementally propose a different task completion as the new task is defined. The result is that not all the steps of the new task need to be provided, and the effort of giving a new task to the robot is reduced.

Our complete generalization and autocompletion contributions are effective in different real robots, including a 2-arm manipulator (Baxter), and a mobile wheeled robot. In the paper, we focus on the illustrations of the robot primitives and task graphs, and results with Baxter. Our approach is general to task-based agents and aims at enabling the desired long-term deployment of robots that accumulate experience.

This introduction section is followed by an overview of related work, sampling the extensive past work on task learning from experience, highlighting our approach for its graph-based generalized representation, and for its relevance and application to autonomous agents. We then introduce the technical components of our approach, present results, and conclude the paper with a review of the contributions and hints of future work.

2 Related Work

Accumulating and reusing tasks has been widely studied in multiple contexts, including variations of case-based planning [9,11,22,4], macro learning [7], and chunk learning at different levels of granularity [13,1,16,14,5]. Tasks, seen as plans, are solved and generated by algorithms that benefit from accumulated experience to significantly reduce the solving effort. Such methods learn based on explanations captured from the dependencies in actions, producing sequential representations. Our approach does not depend on an automated problem solver, as users could be providing the tasks. We represents tasks as graph-based structures with sequences, conditionals, and looping constructs. Generalization and autocompletion use task structures instead of domain-based dependencies.

Multiple techniques address the problem of a robot learning from demonstration [3]. Approaches focus on teaching a single tasks, using varying representations. Tasks have been represented as acyclic graphs composed of nodes representing finite state machines [21] without loops or variables. More recently, complex tasks are represented as Instruction Graphs [15], which only handle instantiated tasks, or Petri Net Plans [8], which support parameterized tasks defined by a user. We introduce Generalized Instruction Graphs, parameterized tasks that are automatically generated from instantiated tasks. Tasks have also been represented as policies that determine state-action mappings [6,2,17,20].

Tasks have been generalized from multiple examples, where each example corresponds to exactly one past task, and the user specifies the generalized class [18]. Our algorithm generalizes and finds parts of tasks patterns that share any graph structure beyond dependency structure [11,22].

To represent conditionals and loops, many task representations are graph-based. Therefore, generalization from examples requires finding common subgraphs between different tasks. The problem of finding labeled subgraph isomorphisms is NP-Hard [12]. However, the problem becomes tractable on trees. To solve it, multiple algorithms have been proposed for mining common frequent subtrees from a set of trees [10]. One, Treeminer, uses equivalence class based extensions to effectively discover frequently embedded subtrees [24]. Instead, GASTON divides the frequent subgraph mining process into path mining, then subtree mining, and finally subgraph mining [19]. We use 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 [23]. SLEUTH uses scope-lists to compute the support of the subtrees, while adopting a class-based extension mechanism for candidate generation. Our mined tree patterns are further filtered out to capture executable and parameterized task graphs.

3 Approach

We consider a robot with modular primitives that represent its action and sensing capabilities. We assume the robot has a library of common tasks, where each task is composed of these primitives. Our goal is to identify common frequent subtasks and generalize over them with limited user assistance. In this section, we first give a brief overview of the graph-based task representation used in this work. Then, we present an in-depth description of the generalization and task autocompletion algorithms.

3.1 Instruction Graphs

The task representation we use is based on Instruction Graphs (IG) [15]. In the IG representation, vertices contain robot-primitives, and edges represent possible transitions between vertices. Mathematically, an Instruction Graph is a graph $G = \langle V, E \rangle$ where each vertex v is a tuple:

$$v = \langle id, InstructionType, Action \rangle$$

where the *Action* is itself a tuple:

$$Action = \langle f, P \rangle$$

where f is a function, with parameters P. The function f represents the action and sensing primitives that the agent can perform. We introduce parameter *types* related to their purpose on the robot. For instance, on a manipulator, the function *set_arm_angle* has parameters of type *Arm* and *Angle*, with valid values of $\{left,right\}$ and $[0, 2\pi]$ respectively. The primitives and parameter types are robot-specific.

Each Instruction Graph is executed starting from an initial vertex, until a termination condition is reached. During execution, the *InstructionType* of the vertex describes how the robot should interpret the output of the function f in order to transition to the next vertex. The IG framework defines the following types:

- *Do and DoUntil*: Used for open-loop and closed-loop actuation primitives. The output of f is ignored as there is only one out-edge. For simplificty, we refer to both of these types as Actions.
- *Conditionals*: Used for sensing actions. The output of f is interpreted as a boolean value used to transition to one of two children.
- *Loops*: Used for looping structures. The output of f is interpreted as a boolean value, and actions inside of the loop are repeated while the condition is true.

Additionally, we use a *Reference InstructionType* for specifying hierarchical tasks.

- *References*: Used to execute Instruction Graphs inside of others. The output of f is interpreted as a reference to the other task.

Figure 1 shows an example node with id 4, *InstructionType* Action, function *set_arm_angle*, and parameters *left* and π. For a more detailed overview of Instruction Graphs, we refer the reader to [15]. However, the generalization algorithms we discuss can be applied to other graph-like representations [8,21].

3.2 Generalizing Tasks

In this section we describe our algorithm to extract generalized tasks from a library of Instruction Graphs. We define a general task as a *Generalized Instruction Graph* (GIG). In a GIG, the parameters of some actions are ungrounded.

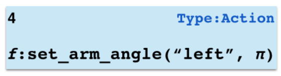

Fig. 1: Example of Instruction Graph node with id 4, *InstructionType* Action, function *set_arm_angle*, and parameters *left* and π.

In such cases, we know the type of these ungrounded parameters, but not their value. So, for each parameter we associate a distribution over all known valid groundings. For instance, in the case of a grounded parameter, the distribution always returns the grounded value. Formally, a GIG is also a graph $GIG = \langle V, E \rangle$ where each vertex v is a tuple:

$$v = \langle id, InstructionType, GeneralAction \rangle$$

$$GeneralAction = \langle f, P, \Phi \rangle$$

where $\phi_i \in \Phi$ is a distribution over groundings of the parameter $p_i \in P$.

These distributions are learned during task generalization and are used to propose initial parameters during task autocompletion. A GIG can be instantiated as an IG by grounding all of the uninstantiated parameters. This process consists of replacing any unspecified p_i with an actual value.

Our approach generates a library of GIGs from a library of IGs, as shown in Algorithm 1. The general problem of finding labeled subgraph isomorphisms is NP-Hard. However our problem can be reformulated into the problem of finding common labeled subtrees in a forest of trees. To this end, we create a tree representation of each IG. As the first step, we define a mapping from IGs to Trees (T):

$$toTree : IG \rightarrow T$$

and its corresponding inverse:

$$toIG : T \rightarrow IG$$

The function *toTree* computes a labeled spanning tree of an input Instruction Graph (line 3). Specifically, *toTree* creates a spanning tree rooted at the initial vertex of the input IG, by performing a depth first search and by removing back edges in a deterministic manner. This ensures that instances of the same GIG map to the same spanning tree.

Each node in the tree is labeled with the *InstructionType* and function f of the corresponding node in the IG. In this label, we do not include the parameters because we eventually want to generalize over them.

Next, we use a labeled frequent tree mining algorithm to find frequently occurring tree patterns (line 5). A frequently occurring tree pattern is a subtree that appears more than a threshold σ, called the *support*. A tree-mining algorithm *ftm* takes as input a set of trees and the support. As output, it provides a mapping from each tree pattern to the subset of trees that contain it. Then, since each tree pattern is associated to a set of trees and each tree corresponds

Algorithm 1 Task Generalization

```
 1: procedure GENERALIZETASKS($IGs$, $\sigma$, $L$)
 2:   // IG library is converted to trees
 3:     $IGTrees \leftarrow \{toTree(g) \mid g \in IGs\}$
 4:   // Tree patterns are found by a tree mining algorithm
 5:     $tp \leftarrow \mathrm{ftm}(IGTrees, \sigma)$
 6:   // Mapping from tree patterns to IGs is created
 7:     $igp \leftarrow \{\langle p,\ toIG(T)\rangle \mid \langle p, T\rangle \in tp\}$
 8:   // Filters remove unwanted tree patterns
 9:     $igp \leftarrow \mathrm{filter\_not\_exec}(igp)$
10:     $igp \leftarrow \mathrm{filter\_by\_length}(igp, L)$
11:   // Tree patterns of full tasks are reintroduced
12:     $igp \leftarrow \mathrm{add\_full\_igs}(IGs, igp)$
13:   // Vertices and edges of the GIGs are constructed
14:     $gigs \leftarrow \mathrm{create\_ugigs}(IGs, igp)$
15:   // Parameters and distributions are computed
16:     $gigs \leftarrow \mathrm{parametrize}(IGs, igp, gigs)$
17:     return $gigs$
18: end procedure
```

to a specific IG, we can create a mapping directly from tree patterns to IGs (line 7). We denote this mapping as *IGP*.

A tree mining algorithm will return many tree patterns. In particular, for any tree pattern, any subtree of it will be returned. This is because each subtree will have a support at least as large as its parent. Rather than keeping all these patterns, we focus on storing those that are the most applicable. There are many possible ways to filter the patterns. We propose several heuristic filters that select patterns based on their *executability*, *frequency*, and *usefulness*.

- **Executable** patterns are those that the robot can run.
- **Frequent** patterns are statistically likely to appear in the future.
- **Useful** patterns reduce many interactions when correctly proposed.

Each filter is formally defined as a function:

$$\mathit{filter} : IGP \rightarrow IGP$$

We first filter patterns that cannot be executed by the robot. In particular, we remove patterns with incomplete conditionals and loops (line 9).

Then, there is a tradeoff between highly frequent patterns and highly useful patterns. Patterns that occur with a large frequency are typically smaller, so they provide less utility during task autocompletion. Larger patterns provide a lot of utility, but they are usually very specific and occur rarely.

Less frequent patterns are already filtered out by the tree-mining algorithm when we provide it a minimum support σ. To optimize for larger patterns that save more steps during autocompletion, we also remove patterns that are shorter than a threshold length L (line 10). This ensures that we do not keep any pattern that is too small to justify a recommendation to the user.

Finally, since we are dealing with autocompletion for robots, one desirable feature is to be able to propose entire tasks. Even if a full task has a low support, or is below the threshold length, there is value in being able to propose it if a reparameterized copy is being provided to the robot. Consequently, we reintroduce the tree patterns corresponding to full IGs (line 12).

Even with these filters, we still keep some tree patterns that are complete subtrees of another pattern. In practice, many of these patterns provide useful task autocompletion suggestions. However, in memory-limited systems, we suggest also filtering them.

Finally, the algorithm processes the filtered set of tree patterns to create GIGs by creating vertices and edges from the tree pattern and then parameterizing the vertices. First, to create the GIG's vertices and edges, we copy the subgraph corresponding to the tree pattern from any of the IGs containing the pattern (line 14). This gives us a completely unparameterized GIG (*uGIG*), with no parameter distributions. Next, we determine which parameters are grounded in the GIG, and which are left ungrounded. A parameter is instantiated if it occurs with the same value, with a frequency above a given threshold, in all corresponding IGs. Otherwise, the parameter is left ungrounded with an empirical distribution.

This process is repeated for every subtree pattern not removed by our heuristic filters, creating a library of GIGs (line 16). When this algorithm is run incrementally, this library is unioned with the previous library.

Figures 2a and 2c show example IGs for a task that picks up an object, and drops it at one of two locations. Figures 2b and 2d depict their corresponding spanning trees. Finally, Figure 3 shows the general task that is extracted. In this GIG, the parameters in nodes 3 and 4 are kept instantiated, since they were shared by the two original IGs. The others parameters instead are left ungrounded. These parameters have a type *id*, and a distribution over the landmark ids $\{1, 3\}$, which were extracted from the IGs in Figure 2.

3.3 Task Autocompletion

We now consider an agent that is provided a task incrementally through a series of interactions. Each interaction consists of adding a vertex to the graph or modifying an existing vertex. At any step of this process, the agent knows a *partial task*. After each interaction, this partial task is compared against the library of GIGs to measure task similarity and perform autocompletion.

The algorithm performs this comparison by checking if the end of the partial task being provided is similar to a GIG (Algorithm 2). Specifically, we keep a set of candidate proposals, denoted *props*, that match the final part of the partial task. When the partial task changes (line 2), we first update this set to remove any elements that no longer match the task being taught (line 3). Then, we add new elements for every GIG that starts with the new vertex (line 4). When a threshold percentage τ of one or more GIGs in this set matches

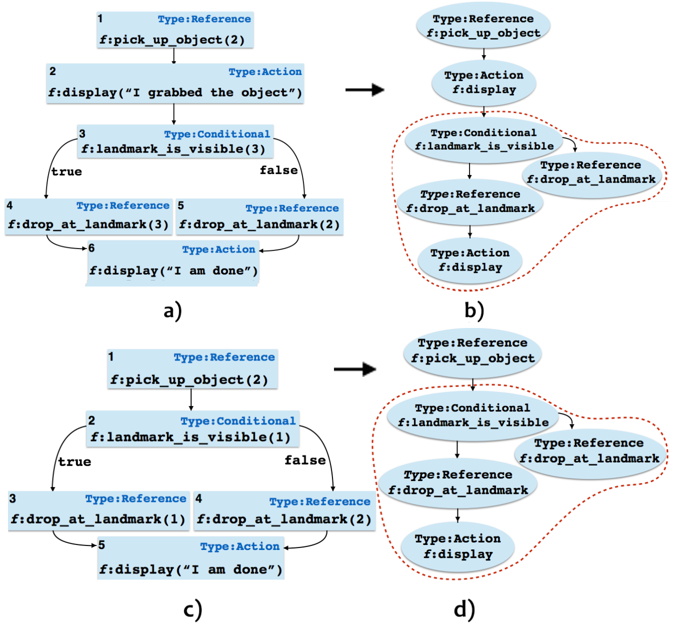

Fig. 2: Example of two Instruction Graphs (a, c) converted into their corresponding spanning trees (b, d). The tree pattern shared between them is circled in red.

the current partial task, the robot proposes the largest GIG and breaks ties randomly (lines 6 and 7).

When a specific proposal is found, the robot displays a representation of the GIG and asks for permission to demonstrate the task. Having previously filtered all the incomplete GIGs, all the proposals can in fact be executed. When granted permission, the agent demonstrates an instance of the GIG, noting when a parameter is ungrounded.

At the end of the demonstration, the agent asks if the partial task should be autocompleted with the demonstrated task. If so, the agent asks for specific values for all of the ungrounded parameters. At this stage, the agent suggests initial values for each ungrounded parameter p_i by sampling from its corresponding distribution ϕ_i.

After all of the parameters are specified, the nodes matching the general task in the partial task are replaced with one *Reference* node. When visited, this node executes the referenced GIG, instantiated with the provided parameters. With this substitution, the length of the task is reduced.

Figure 4 shows a sample task acquisition for a Baxter manipulator interacting with a user. After the first command, the robot finds at least one general task starting with *display_message*. However, none of the GIGs recognized surpass the similarity-threshold τ. When the third instruction is given to the agent, this threshold is surpassed, and the autocompletion procedure is started. First, the

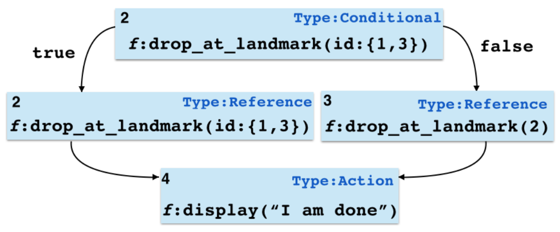

Fig. 3: Example GIG that is extracted from the graphs in Figures 2a and 2c. The parameters in nodes 3 and 4 are instantiated, since they were shared by the two original IGs. Instead, the parameters in nodes 1 and 2 are left ungrounded.

Algorithm 2 Task Autocompletion

```
 1: procedure AUTOCOMPLETE($GIGs, ig, props, \tau$)
 2:     if $hasChanged(ig)$ then
 3:         $props \leftarrow$ deleteNotMatching($ig, props$)
 4:         $props \leftarrow$ addNewMatching($ig, props, GIGs$)
 5:         ($best, similarity$) $\leftarrow$ bestMatch($ig, props$)
 6:         if $similarity \geq \tau$ then
 7:             propose($best$)
 8:         end if
 9:     end if
10: end procedure
```

robot asks permission to perform a demonstration of the general task. After completing the demonstration, the robot asks if the autocompletion is correct. If so, it also asks for ungrounded parameters to be specified and suggests values using each parameter's distribution.

4 Experiments

Several of our robots can perform generalization and autocompletion, including a manipulator and a mobile base. In order to demonstrate the value of our approach, we define two sets of tasks. Intuitively, the first set of tasks represents a robot that is still acquiring completely new capabilities. Instead, the second set of tasks represents a robot that is acquiring instances of tasks that it already knows. More formally, in the first set, S_d no tasks are repeated. They share only a small fraction of similar components that can be generalized. To show that generalization takes place, we use a second set, S_r consisting of two repetitions of each of elements in S_d with different parameters. We see that the algorithm recognizes and autocompletes the second instance of each task.

An additional benefit of this approach is that we we can keep one common library for all of our robots. If the robots have different primitives, their tasks are automatically generalized apart. However, if this share primitives, our approach can learn subgraphs common to both robots. One concern is that this library

```
Example Interaction

U: Open Gripper
R: I will open my gripper.
R: What should I do next?
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: GIG_14.
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 90 degrees (open).
R: This is my full suggestion.
R: Would you like to use it?
U: Yes.
[User specifies open parameters]
[User can rename the GIG]
```

Fig. 4: Sample autocompletion interaction during task acquisition.

could be very large. We have accumulated libraries of up to 10,000 tasks, by creating parametric variations of a smaller core of tasks. Even on libraries this large our approach runs in under 4 seconds.

In the rest of this section, we show in detail experiments run on a Baxter manipulator robot.

4.1 Experiments with Baxter

Baxter has two 7 degree of freedom arms, cameras on both arms, and a mounted Microsoft Kinect. The robot-primitives on Baxter manipulate the arms, open their grippers, display messages, and sense landmarks. The frequent subtree mining algorithm we employ is an open source version of SLEUTH [1].

The tasks that Baxter can perform range from waving to making semaphore signs to pointing at landmarks. Many of Baxter's tasks involve picking up an object and moving it to another location. For instance, Figure 5 shows Baxter searching for a landmark to see if a location is unobstructed to drop a block. Without task generalization and autocompletion, a new task must be provided for each starting location and ending location in Baxter's workspace. With generalization and autocompletion, these locations become ungrounded parameters that can be instantiated with any value.

For this experiment, 15 tasks were taught by two users familiar with robots but not the teaching framework. These tasks ranged from waving in a direction,

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

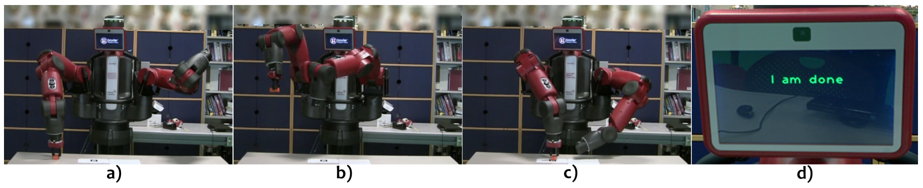

Fig. 5: Baxter performing an instance of the GIG shown in Figure 3. First, Baxter picks up the orange block (a); then Baxter checks if a location is unobstructed with its left arm camera (b); Since the location is unoccupied, Baxter drops the Block. (c); Finally, Baxter says that it is done (d).

to pointing to visual landmarks, to placing blocks at different positions, to performing a series of semaphore signals to deliver a message. S_d has 15 distinct tasks and S_r has 30 tasks. The average length of a task in both sets is 9.33 nodes.

4.2 Experimental Results

As we accumulate the library incrementally, the order in which tasks are provided affects the generalization. To account for this, we ran 1000 trials where we picked a random ordering to our sets and had a program incrementally provide them to Baxter. The GIG library was updated after each task was provided. At the end of every trial, we measured the number of steps saved using autocompletion compared to providing every step of the task. This measurement includes steps added due to incorrect autocompletion suggestions. For this particular experiment, the support was fixed to 2, the minimum GIG length to 2, and task autocompletes were suggested if $\tau = 30\%$ of a GIG matched the partial task.

We compare our results to an near-upper bound on the number of steps saved by an optimal autocompletion algorithm. For any set of tasks that take n steps to be provided, and are only proposed once τ percent of a task matches a GIG, we have:

$$OPT \leq (1 - \tau) \cdot n$$

This corresponds to perfectly generalizing every task after τ percent of it has been acquired. We note that an algorithm could do slightly better by also making smaller proposals for the first τ percent of each task. For S_d and S_r we have:

$$OPT_d \leq 92$$

$$OPT_r \leq 184$$

Table 1 reports the result of the experiment. Specifically, in this table we report the following measures:

- **Maximum Steps Saved (%):** maximum percentage of steps saved over all permutations, in comparison to the theoretical upper bound.
- **Average Steps Saved (%):** average percentage of steps saved over 1000 permutations, in comparison to the theoretical upper bound.

– **Average Partially Autocompleted:** average number of tasks that were *partially* autocompleted with a GIG.
– **Average Completely Autocompleted:** average number of tasks that were *completely* autocompleted with a GIG.

Table 1: Results obtained for the two sets taught to the Baxter robot.

	1st Set (S_d)	2nd Set (S_r)
Max. Steps Saved	70.65%	100%
Avg. Steps Saved	$33.44 \pm 14\%$	$81.92 \pm 7.05\%$
Part. Autocompleted	4.72 ± 1.56	4.70 ± 1.58
Compl. Autocompleted	0 ± 0	15 ± 0

As expected, S_r benefits the most from the autocompletion method. Specifically, the average percentage of steps saved compared to OPT_r is approximately 82%. For S_d, the average compared to OPT_d is 33%. In the former case, the robot is able to leverage the knowledge of the similar tasks it already knows. Indeed, our approach meets the theoretical upper bound when provided tasks from S_r in the optimal ordering. This fact is additionally underlined by the number of tasks in which the robot suggested any correct GIG. In particular, on average the robot proposed a correct autocompletion suggestions for 65% of graphs in the second set, and 30% in the first set.

Also, the 15 IGs added to the second set are all completely autocompleted from their other similar instance. Furthermore, this happens with a statistically insignificant change to the effectiveness of the partial autocompletions.

Finally, the size of the GIG library for the first set was 21 and that the size of the GIG library for the second set was 45. This shows that the heuristic filters we proposed achieves a good balance between saving steps and library size.

5 Conclusion and Future Work

In this paper, we considered autonomous robots that persist over time and the problem of providing them additional tasks incrementally. To this end, we contributed an approach to generalize graph-based tasks, and an algorithm that enables the autocompletion of partially specified tasks.

Our generalization and autocompletion algorithms have been successfully deployed on multiple robots, acquiring large task libraries. Our experiments report in-detail the effectiveness of our contributions on a Baxter manipulator robot for two sets of tasks. With both sets, we found a significant reduction in the number of steps needed for Baxter to acquire the tasks.

In terms of future work, there may be other applicable filters for deciding which tree patterns should be converted to GIGs. Furthermore, structure-based generalization is just one way for a robot to express its capabilities. Future research may look at domain-specific forms of task generalization.

References

1. Anderson, J.R., Bothell, D., Lebiere, C., Matessa, M.: An integrated theory of list memory. Journal of Memory and Language (1998)
2. Argall, B.D., Browning, B., Veloso, M.: Learning robot motion control with demonstration and advice-operators. In: IROS (2008)
3. Argall, B.D., Chernova, S., Veloso, M., Browning, B.: A survey of robot learning from demonstration. Robotics and autonomous systems (2009)
4. Borrajo, D., Roubíčková, A., Serina, I.: Progress in case-based planning. ACM Computing Surveys (CSUR) 47(2), 35 (2015)
5. Borrajo, D., Veloso, M.: Lazy incremental learning of control knowledge for efficiently obtaining quality plans. In: Lazy learning (1997)
6. Chernova, S., Veloso, M.: Learning equivalent action choices from demonstration. In: IROS (2008)
7. Fikes, R.E., Hart, P.E., Nilsson, N.J.: Learning and executing generalized robot plans. Artificial intelligence (1972)
8. Gemignani, G., Bastianelli, E., Nardi, D.: Teaching robots parametrized executable plans through spoken interaction. In: Proc. of AAMAS (2015)
9. Hammond, K.J.: Chef: A model of case-based planning. In: AAAI (1986)
10. Jiang, C., Coenen, F., Zito, M.: A survey of frequent subgraph mining algorithms. The Knowledge Engineering Review (2013)
11. Kambhampati, S.: A theory of plan modification. In: AAAI (1990)
12. Kimelfeld, B., Kolaitis, P.G.: The complexity of mining maximal frequent subgraphs. In: Proc. of the 32nd Symp. on Principles of Database Systems (2013)
13. Laird, J.E., Rosenbloom, P.S., Newell, A.: Chunking in soar: The anatomy of a general learning mechanism. Machine learning (1986)
14. Langley, P., McKusick, K.B., Allen, J.A., Iba, W.F., Thompson, K.: A design for the icarus architecture. ACM SIGART Bulletin (1991)
15. Meriçli, Ç., Klee, S.D., Paparian, J., Veloso, M.: An interactive approach for situated task specification through verbal instructions. In: Proc. of AAMAS (2014)
16. Minton, S.: Quantitative results concerning the utility of explanation-based learning. Artificial Intelligence (1990)
17. Ng, A.Y., Russell, S.J., et al.: Algorithms for inverse reinforcement learning. In: ICML (2000)
18. Nicolescu, M.N., Matarić, M.J.: Natural methods for robot task learning: Instructive demonstrations, generalization and practice. In: Proc. of AAMAS (2003)
19. Nijssen, S., Kok, J.N.: The gaston tool for frequent subgraph mining. Electronic Notes in Theoretical Computer Science (2005)
20. Ratliff, N.D., Bagnell, J.A., Zinkevich, M.A.: Maximum margin planning. In: Proc. of the 23rd international conference on Machine learning (2006)
21. Rybski, P.E., Yoon, K., Stolarz, J., Veloso, M.: Interactive robot task training through dialog and demonstration. In: Proceedings of the 2th ACM/IEEE International Conference on Human-Robot Interaction (2007)
22. Veloso, M.M.: Planning and learning by analogical reasoning. Springer Science & Business Media (1994)
23. Zaki, M.J.: Efficiently mining frequent embedded unordered trees. Fundamenta Informaticae (2005)
24. Zaki, M.J.: Efficiently mining frequent trees in a forest: Algorithms and applications. IEEE Transactions on Knowledge and Data Engineering (2005)