What Were You Thinking? Filling in Missing Dataflow Through Inference in Learning from Demonstration

Melinda T. Gervasio
SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025, USA
Tel: 1-650-859-4411
E-mail: melinda.gervasio@sri.com

Janet L. Murdock
SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025, USA
Tel: 1-650-859-4860
E-mail: janet.murdock@sri.com

ABSTRACT
Recent years have seen a resurgence of interest in programming by demonstration. As end users have become increasingly sophisticated, computer and artificial intelligence technology has also matured, making it feasible for end users to teach long, complex procedures. This paper addresses the problem of learning from demonstrations involving unobservable (e.g., mental) actions. We explore the use of knowledge base inference to complete missing dataflow and investigate the approach in the context of the CALO cognitive personal desktop assistant. We experiment with the Pathfinder utility, which efficiently finds all the relationships between any two objects in the CALO knowledge base. Pathfinder often returns too many paths to present to the user and its default shortest path heuristic sometimes fails to identify the correct path. We develop a set of filtering techniques for narrowing down the results returned by Pathfinder and present experimental results showing that these techniques effectively reduce the alternative paths to a small, meaningful set suitable for presentation to a user.

ACM Classification: I.2.6 [Artificial Intelligence] Learning: Induction, Knowledge acquisition; I.2.8 [Artificial Intelligence] Problem Solving, Control Methods, and Search; I.2.3 [Artificial Intelligence] Deduction and Theorem Proving.

General terms: Algorithms, Performance, Experimentation, Human Factors

Keywords: Programming by demonstration, programming by example, task learning, end-user programming, knowledge base inference

INTRODUCTION
Programming by demonstration (PBD) [9,15] has seen a resurgence in recent years as the personal computer has become more powerful and end users have become increasingly sophisticated, becoming familiar with and often quite skillful in the use of myriad desktop and Web applications. As the tools have become more powerful, users have started using them to accomplish more and more complex tasks. Supporting end users in the task of teaching a system such procedures by demonstration has thus resulted in the development of various approaches. On one hand are tools to simplify the task of "programming" for the end user. For example, Tailor [4] assists the user in procedure construction and modification by mapping user instructions to corresponding procedure steps and modification templates, while Koala [16] relies on *sloppy programming* to represent procedures in a way that is understandable to users while also being translatable into executable code. On the other hand is the integration with more reasoning technology to enable the learning system to learn more from a single example. For example, POIROT [6] uses plan recognition to drive top-down learning and complement the system's bottom-up empirical learners, while LAPDOG [12] uses knowledge base inference to fill in missing dataflow. In between are approaches that cast the task learning process as a collaborative one between user and system. For example, Collagen [11] lets users annotate their demonstrations to provide additional information to the system, while PLOW [1] leverages the user's commentary during demonstration to help in identifying procedure parameters and procedure structure.

In our work on LAPDOG [12], we have strived to develop techniques to learn from demonstration on the computer desktop across a variety of applications. The goal of having appropriately instrumented applications has always been a primary challenge for PBD and most previous work [9,15] has focused on single, often proprietary, applications. More recently, there has been a lot of focus on the Web (e.g., [1,5,6,16]), since the instrumentation of a single "application" (a Web browser) immediately provides access to a wealth of different services. LAPDOG takes advantage of the integrated CALO desktop environment [7], not just for the instrumentation and automation framework but also for the centralized knowledge base that provides a shared in-

IUI'09, February 8–11, 2009, Sanibel Island, Florida, USA.

formation repository for the different components of the CALO cognitive assistant.

In this paper, we present our work on dataflow completion through knowledge base inference. In an effort to let users demonstrate procedures more naturally, we address the issue of learning from demonstrations that are incomplete due to certain unobservable actions by the user. The technique we describe and explore in this paper involves using inference to fill in these blanks and learn a valid procedure. We begin by describing the CALO framework that supports task learning across a variety of applications as well as reasoning about the office environment. We then present LAPDOG, describing the missing dataflow problem and our approach to dataflow completion. We follow this with a presentation of Pathfinder, the knowledge base (KB) component that LAPDOG uses in dataflow completion to find possible relations between objects. In exploratory experiments with Pathfinder, we discovered that its default, shortest path heuristic sometimes fails to identify the correct path. We present the set of filtering techniques we devised to address this problem and then describe our evaluation of Pathfinder and the effects of our filtering techniques. We conclude with a discussion of the results and directions for future work.

CALO

CALO [7] is an adaptive, cognitive, personalized assistant designed to assist office workers, primarily in their activities on the electronic desktop. CALO provides numerous assistant capabilities across standard applications on the Windows platform, backed by various learning and reasoning modules to support information organization, information and task prioritization, and task management. A centralized knowledge base serves both as a source of background knowledge and as a target for the information generated by a collection of engineered harvesters and learned extractors and classifiers.

CALO Ontology

The CALO Ontology is a collection of specialized ontologies designed to capture the knowledge requirements of an adaptive, cognitive personal assistant [8]. For the purpose of this paper, there are two subontologies of interest: the Core+Office Ontology and the Task Ontology. The *Core Ontology* or upper ontology is CLib [2], a set of reusable, composable domain-independent components for building knowledge representations. The *Office Ontology* is a specific instantiation developed for the CALO office domain and captures entities such as *EmailMessage*, *ComputerFile*, and *CalendarEntry*; roles such as *ProjectLeaderRole, AdministrativeAssistantRole,* and *EmployeeRole*; and (invertible) relations such as *authorOf* (*authorIs*), *calendarEntryAttendeeOf* (*calendarEntryAttendeeIs*), and *nextEmailInThreadOf* (*nextEmailInThreadIs*). For example, Figure 1 presents a graphical depiction of the knowledge that Karen is the leader of the PExA project of which Melinda and Janet are participants.

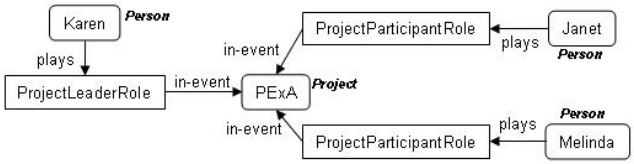

Figure 1. CALO knowledge base connections between a project leader and project participants.

The *Task Ontology* defines the tasks or actions that CALO can observe and/or perform. It includes definitions for actions a user might perform on various desktop applications (e.g., *SendEmail*, *OpenMSOfficeDocument*, and *OpenUrl*), CALO processes the user might invoke (e.g., *ScheduleMeeting*), and actions CALO might perform in the background (e.g., *HarvestEmail*, *PredictFolder*, *ExtractFileSummary*). Tasks operate on CALO Ontology entities; Figure 2 partially illustrates the *SendEmail* task.

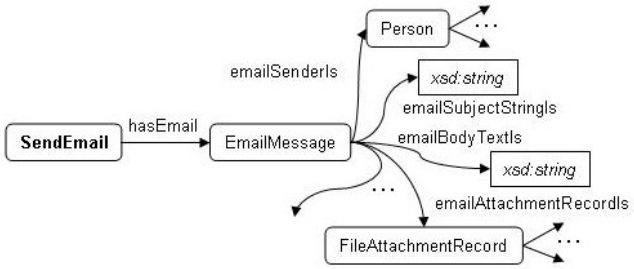

Figure 2. Partial depiction of the SendEmail task.

Capturing User Actions on the CALO Desktop

CALO's instrumentation framework was designed to capture actions corresponding to application-level user operations rather than state changes, the idea being to represent actions as users tend to think of them when they use an application. For example, users typically think of opening and closing documents, reading and sending email, and filling in form fields rather than pulling down menus and clicking on menu items or typing individual characters. The tradeoff is that while we can observe actions across a wide variety of applications, we cannot observe finer-grained actions within individual applications. For example, we can observe an email composition window being opened and attachments being added to it but we cannot observe the individual keystrokes as the user writes the message.

Capturing actions at the application level results in a procedure representation that is cognitively meaningful to users, providing the opportunity to more directly engage the user in the development and modification of procedures. Another consequence of this approach to instrumentation is that mapping observations to actions becomes straightforward. Unlike in other PBD systems where instrumentation captures state and the learning task is to induce the actions whose execution will result in the observed state changes, LAPDOG's primary learning task is to generalize the dataflow and the procedural structure seen in a demonstration.

All the instrumentation events flow through the CALO publish-and-subscribe mechanism in the form of instances of the tasks defined in the Task Ontology. Thus, from LAPDOG's perspective, there is no particular difference between an action from a Web browser and an action from an email application or an action on the desktop itself, and learning procedures across disparate applications comes naturally. However, this unified framework comes at the cost of a not insignificant burden on the instrumentation. Thus, while we are able to get instrumentation from a larger set of applications, we get fewer instrumentation events from each one, making it even more likely that demonstrations will be incomplete because of certain unobservable actions.

LAPDOG

LAPDOG (Learning Assistant Procedures from Demonstration, Observation, and Generalization) is one of the several task learning components in CALO. It provides the capability of learning procedures from demonstration while other modules provide complementary functionalities such as primitive action acquisition [5], procedure editing [4], and procedure execution [17]. We begin by laying out some terminology before presenting LAPDOG's approach to learning from demonstrations involving unobservable mental actions.

Terminology

An *action* is defined by a set of input and output arguments. Let $A(+i_1 \ldots +i_m -o_1 \ldots -o_n)$ denote the action A with *input arguments* $\{i_1, \ldots, i_m\}$ and *output arguments* $\{o_1, \ldots, o_n\}$. An argument has an associated semantic type and may be a scalar, a set, a list, or a tuple (i.e., a record or heterogeneous fixed-size list). Sets, lists, and tuples may be of arbitrary depth.

A *demonstration* consists of a sequence of actions with constant arguments. The dataflow paradigm requires ensuring that every input of every action is supported by some output or outputs of previous actions. Currently in LAPDOG, a scalar input may be supported by a previous scalar output of the same value, or by a list or tuple accessor expression over a previous list or tuple output that yields the same value. Meanwhile, a list input may be supported by a previous list output of the same value or by a list construction operation over previous outputs that yields the same value. LAPDOG does not yet support set or tuple construction expressions and so a set or tuple input can be supported only by a previous set or tuple of the same value. However, these extensions should be fairly straightforward.

Learning from Demonstrations with Missing Dataflow

Given a demonstration, LAPDOG's task is to generalize this into an executable procedure. This involves two major operations: dataflow completion and generalization. As a side effect, LAPDOG also determines the *task signature*—specifically, the inputs required by the procedure and the outputs generated by its execution. We now describe each of these steps in turn.

Dataflow Completion. For a procedure to be executable, every input of every action must be supported. Because demonstrations may include unobservable actions (e.g., mental operations by the user), the dataflow in the observed actions may be incomplete (i.e., certain inputs may be unsupported). LAPDOG thus attempts to find additional actions that could support these unsupported inputs and complete the dataflow.

There are two ways LAPDOG can complete a dataflow. The first is through the insertion of *information-producing actions* to generate required inputs from known outputs. LAPDOG determines these actions by running a depth-bounded forward search over the space of information-producing actions, essentially creating an *information plan* for generating the required inputs. Examples of information-producing actions are information extractors and string manipulation operations (see [12] for some examples of their use). The list construction and list/tuple accessor operations discussed in the next section may also be viewed as information-producing actions. However, the specialized context of their use makes their consideration during generalization a more appropriate method for incorporating them into the learned procedure.

The second method for completing a dataflow is through a KB query to follow a relational path from known outputs to required inputs. This involves the use of Pathfinder, a KB utility for finding connections between objects in the KB. This method of dataflow completion is the focus of this paper and we discuss it in greater detail in a later section.

Generalization. Once the dataflow is completed, LAPDOG performs two types of generalization: parameter generalization and structural generalization. Parameter generalization involves finding the possible supports for an input and recording each support by a common variable or by a list/tuple expression over variables that unifies the supporting output(s) with the supported input. Structural generalization involves inducing looping structures over the action sequence. Some notable features of LAPDOG's loop learning are the ability to induce loops over sets or lists, loops over multiple lists, and loops that generate (output) lists. Further details about LAPDOG's generalization capabilities can be found in [10].

User Interaction

Figure 3 shows a sample demonstration, as observed by LAPDOG, where the user performed a web search on a particular phrase and then visited the resulting URL. After learning, LAPDOG presents the learned procedure to the user for verification (Figure 4). In this clarification window, LAPDOG presents the learned procedure together with whether parameters are considered constants ("Always use this") or variables ("Ask me for this"), or generated at runtime by some previous action ("Deduce value"). Should the user modify any of these parameter settings or provide additional procedure inputs ("Add variable"), LAPDOG takes the new input and adjusts the learned procedure.

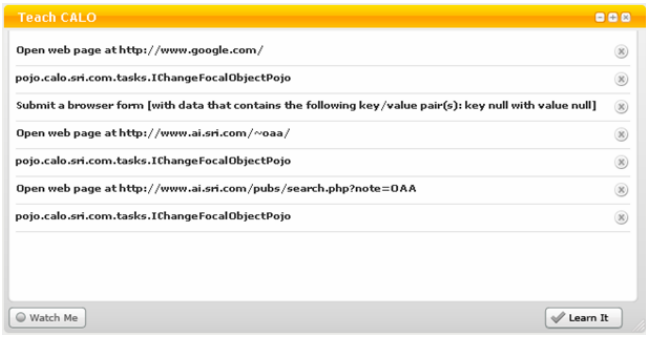

Figure 3. Sample observed demonstration.

The final learned procedure is registered into the CALO Task Ontology and becomes available for invocation. The invocation of a procedure itself generates an instrumentation event and by virtue of its connection to the CALO ontology, it becomes available for use in subsequent demonstrations, providing LAPDOG with the ability to learn over learned procedures.

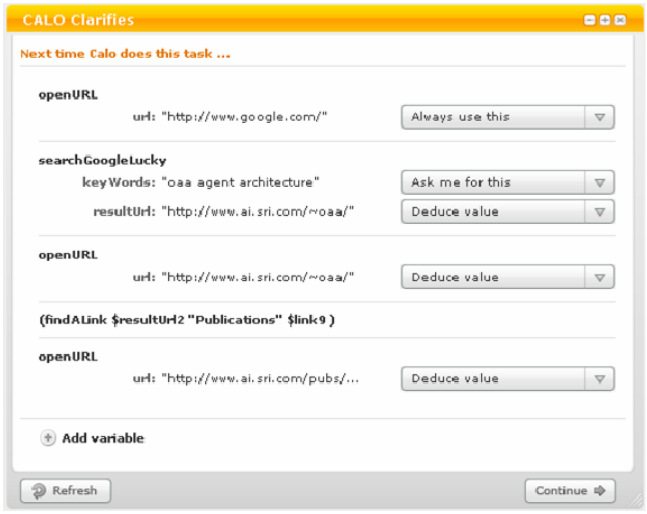

Figure 4. Clarification window for presenting a learned procedure and letting the user make adjustments to parameter settings.

PATHFINDER

A primary method for dataflow completion in LAPDOG is finding relational paths in the CALO KB. Given a set of known outputs and an unsupported input, a KB query that relates any of the known outputs to the input will complete the dataflow. However, given that CALO is designed to be an intelligent cognitive assistant not just for the electronic desktop but, more generally, for the office domain, the CALO KB comprises a very large, richly connected set of objects. Thus, searching on demand for connections between KB objects is infeasible.

The Query/Update Manager *Pathfinder* utility was developed to support dataflow completion in LAPDOG. The basic idea behind Pathfinder is to precompute the populated paths between objects of certain types to greatly reduce the search space and improve efficiency when finding relations between objects during dataflow completion. After an initial ramp-up period, new objects and relations will not be added very frequently to the KB. Pathfinder takes advantage of this relative stability to periodically crawl the KB to update its record of populated paths. It stores this information in an *A(k)* index [14], a highly efficient indexing structure for answering path queries over graph-structured data.

A complicating factor is the directed nature of the relations in the KB—for example, in a supervisory relation, one person is the supervisor of the other. Considering only the primary direction prevents certain paths from being found (e.g., consider the relation between Karen and Janet in Figure 1).

Since the KB does contain the inverse of every relation, we could simply add the inverse relations to the search space, allowing paths to be found in either direction. However, their blanket inclusion greatly degrades performance—a problem we discuss in the section on future work. Pathfinder currently supports bidirectional paths only in a highly restricted situation. In the case where the source is a string, Pathfinder considers single inverse relations to find the possible objects related to that string. For example, the string "Steven Lee" may be the full name of a Person, the author of an Article, or even the subject of an EmailMessage. Pathfinder then proceeds to find paths between any of the candidate objects and the target. While limited, this ability significantly extends LAPDOG's ability to complete missing dataflow through KB connections since it is often the case that based on the demonstration, sources will be strings (e.g., a person's full name, a project name, a paper title) rather than the corresponding KB objects.

Given a source object, a target object, and a maximum search depth, Pathfinder returns all possible paths between the source and target object up to a maximum search depth, ordered according to path length. An increase in search depth in Pathfinder corresponds to the addition of a pair of predicates *(Class* $?x_i$*)* and *(Relation* $?x_1$ $?x_2$*)*. With the possible addition of a single inverse relation in restricted bidirectional search, a search depth of *m* thus corresponds to a maximum path length of *2m+1*. In our evaluation, we set the maximum search depth to 7 (corresponding to a maximum path length of 15), which was 1 more than the search depth required to generate the longest correct path among our use cases.

Pathfinder Issues

Although the number of paths returned by Pathfinder generally increases exponentially with the maximum path length, we expected that we would be able to cap the maximum path length at a reasonable value that would limit the number of possible paths returned by Pathfinder. The rationale was that users were unlikely to perform long sequences of unobservable actions. The idea behind dataflow completion is to simply fill in those small gaps left by reasonably detailed instrumentation. We also expected that the heuristic of shortest path would often yield the desired solution or that we could at least choose a small set of shortest

paths to present to the user as alternative completions from which to choose.

However, it soon became evident that while these assumptions held in many cases, they did not do so in others. When the maximum path length was small (e.g., 5), Pathfinder returned a reasonable number of alternatives (typically 3 to 5). One can easily imagine presenting these few options to the user. However, there were cases where the correct path was of length 13; in these cases Pathfinder could return hundreds of possible paths. Longer paths are also more difficult to read than shorter paths, so presenting many long paths to the user becomes even less desirable. In these situations, it was also sometimes the case that the shortest path was not the correct one. An additional issue was that even though the relational path returned by Pathfinder was guaranteed to connect the source to the target; the target was not guaranteed to be the unique node connected to the source by this path. For example, while Ray might be related to Karen by virtue of being her supervisor, Ray also supervises many other people. While we could potentially ask the user during execution to choose the correct target from all the values returned by the query, once again we were faced with the situation where there were often a prohibitively large number of these.

Improving Pathfinder Results

While modifying Pathfinder itself to return fewer, better results was a possible approach, we were not in a position to do so; as far as LAPDOG is concerned, Pathfinder is essentially a black-box, third-party application. Furthermore, it was not clear that modifying Pathfinder either to limit the extent of its indexing structure or to modify its use was the right approach. For customers other than LAPDOG, returning all possible paths, regardless of length or number, may be the right thing to do. We thus set out to develop a set of techniques for filtering the results returned by Pathfinder into a smaller, more manageable set. The goal was to reduce the set of candidates to a small enough number to make it feasible to present them to the user during learning and ask for help in selecting the correct path.

Prefer Paths with Fewer Targets. As discussed previously, a path connecting a source to a target may also connect the source to other targets. Intuitively, a path that connects the source to fewer targets is more informative than one that connects the source to more. In the best case, the path is unique to the source and intended target.

Let $T = Query(s,P)$ denote the fact that the relational path P from the source s yields the set of targets T. Given two paths P_1 and P_2 such that $T_1 = Query(s,P_1)$, $T_2 = Query(s,P_2)$, and $|T_1| < |T_2|$, we prefer the more informative path P_1 with the smaller corresponding target set T_1.

Remove Redundant Paths. Because the CALO Ontology comprises a collection of subontologies designed for specific purposes [8], it may include relations that are effectively duplicates of each other. For example, within email threads, the *nextEmailInThread* relation is redundantly represented by the *next-element* relation. While Pathfinder could consider simply ignoring certain relations in favor of others, doing so in a type-independent manner is problematic since certain relations may exist only for certain objects. For example, in non-email contexts, *nextEmailInThread* will not exist, whereas *next-element* may.

We investigate another technique for path filtering that removes these redundant paths. Specifically, given two paths $P_1 = QR_1S$ and $P_2 = QR_2S$ that differ only in subpaths R_1 and R_2 where R_1 and R_2 are equivalent, we retain only one of P_1 and P_2.

Remove Repeated Subpaths. Another issue related to redundant paths is that of repeated subpaths, which can occur when the objects on each end of the subpath are of the same type. For example, in our initial explorations, Pathfinder often returned paths containing increasingly long subpaths of the form *((EmailMessage* $?x_1$*) (nextEmailInThread* $?x_1$ $?x_2$*) (EmailMessage* $?x_2$*) (nextEmailInThread* $?x_2$ $?x_3$*) ... (nextEmailInThread* $?x_{n-1}$ $?x_n$*) (EmailMessage* $?x_n$*))*. This was, in this case, in the context of finding paths between two people who happened to have exchanged email multiple times within a thread. Since the basic relation being captured here is that of one person replying to an email sent by another, the paths involving other email messages in the thread are essentially redundant.

A third technique is thus to identify paths with such repeated subpaths and to eliminate them from consideration. Let $\boldsymbol{P}$ be a family of paths such that each P in $\boldsymbol{P}$ is of the form QR^+S—that is, the paths differ only in how many occurrences of a repeatable subpath R they contain. We retain from $\boldsymbol{P}$ only that minimal path QRS containing exactly one occurrence of R.

Remove Longcuts. Because of the highly connected nature of the CALO KB, there will generally be multiple paths connecting any two objects. Some of these paths are truly distinct and, as discussed earlier, it may be insufficient to simply just choose the shortest one and so we need to consider all of them. However, some paths subsume other paths in the sense that one (longer) path can be seen as taking a detour before getting back on the other (shorter) path. We could not find any justification for ever preferring such *longcuts* and so we consider removing them as well.

The fourth technique for path filtering involves removing such subsumed paths. Specifically, if there exist two paths $P_1 = QS$ and $P_2 = QRS$, we remove P_2 from consideration.

Prefer Common Paths. In the case of a list or set input, LAPDOG requests dataflow-completing paths for each element. For example, in the case of finding support for the recipients of an email message, LAPDOG needs to determine the relation of each recipient to previously known outputs. Since it is highly likely that the members of a collection share a common rationale for being part of the collection used in an action, it seemed reasonable to prefer paths that completed the dataflow for as many members of the collection as possible.

The Query Manager does have facilities that let it aggregate query results so that, for example, one could ask for all the members of a project. However, Pathfinder is constrained

to search the explicitly defined relations in the KB. In the case of project members, each person is individually connected to the project, but there is no entity that captures all the members of the project, much less their email addresses or any common property among them. So completing the dataflow for a collection of objects requires individually completing the dataflow for each one.

The final heuristic for path filtering is thus to prefer common or shared paths when supporting collections. Specifically, given a collection of intended targets $\{o_1,o_2,\ldots,o_n\}$ individually supportable by the sets of paths $\Pi = \{P_1,P_2,\ldots, P_n\}$, we prefer that path P that is a member of as many P_i in Π as possible.

DATA FOR EVALUATION

To evaluate the effectiveness of our path filtering techniques, we needed to find specific use cases for dataflow completion during procedure learning from demonstration. So as not to be limited by the existing instrumentation in CALO, we opted to study the pathfinding problem in isolation. This required finding valid use cases for dataflow completion and CALO KBs against which to test them.

Use Cases for Dataflow Completion

Over the months of June and July in 2008, we conducted an ethnographic study of six users with the broad objective of finding actual tasks that could potentially be learned from demonstration. We had several specific objectives regarding finding use cases for existing and future LAPDOG capabilities, but the specific objective relevant to the Pathfinder evaluation was to find use cases for dataflow completion—that is, procedures that involved unobservable mental actions by the user.

The study involved observing a total of six users (one administrative assistant, one manager, and four computer scientists), each over a period of 1.5 to 2 hours of their choosing. During this period, the users performed repetitive tasks as we conducted a contextual inquiry [3]. The users chose tasks to demonstrate, mostly drawn from their work, but a few were personal tasks. We observed users in their offices using their own computers and applications. Based on the data we gathered at these sessions as well as more detailed analysis later on of videotapes of the sessions, we identified 65 instances of unobservable mental actions, 11 of which we determined could potentially be answered by Pathfinder given the concepts and relations currently represented in the CALO Ontology. We extracted an additional three use cases from preliminary interviews with other subjects regarding their candidate procedures for task automation.

In general, subsets of these 14 use cases were contained within the same scenario (there were seven separate scenarios covering these 14 use cases). For example, one user demonstrated a procedure for sending a quarterly report for a large project of which he was the leader of a smaller subproject. In preparing the report, he needed to find the email request for the quarterly report template he had saved (use case: ComputerFile to EmailMessage), find the meetings he had organized or attended for that project (use case: Project to EventEntry), and find the project participants who had attended those meetings (use case: Project to Person Name). These are examples of unobservable mental actions since the users cannot demonstrate the connection from the opened report template to the email message viewed, from the project input to the meeting viewed, and from the project input to the person name included in an email message.

CALO KBs for Testing Pathfinder

The CALO project [7] involves an annual test designed to measure the effects of learning on performance. One fortuitous side effect of this test is the generation of several fairly well populated CALO KBs. In Year 4, the CALO Test involved 16 subjects using CALO over a *Critical Learning Period* (CLP) of one week (five consecutive days). During this period, the subjects exchanged email, worked collaboratively on documents, scheduled and held meetings, browsed the Web, and conducted various other activities common in an office environment. As a user works, CALO accumulates objects and relations in its KB corresponding to the documents, emails, meetings, etc. accessed by the user. Snapshots (i.e., full backups) of each user's CALO KB were taken at least once a day and always at the end of the day. In our evaluation of Pathfinder, we used the snapshots taken at the end of the fifth day of the CLP.

Instantiating Pathfinder Queries

The goal of this evaluation was to investigate the nature of the answers returned by Pathfinder and the effects of various filtering techniques for narrowing down these answers to a smaller set, suitable for presentation to the user. Thus, we specifically wanted instantiations for which we knew Pathfinder could find the right answer, given that the necessary relational structures were in the KB. For this same reason, we chose to use the six largest CALO KBs from the Y4 Test, presuming that they were likely to contain more objects and richer relationships between the objects.

We eliminated four KBs from consideration because of corrupted data: two because of a particular overzealous learning component (unrelated to task learning) that added hundreds of incorrect relations, one because of missing critical information (project memberships), and one because of multiple sets of redundant objects.

Since we are evaluating Pathfinder in isolation—i.e., independent of an actual instance of learning from demonstration—we needed to address two issues in order to formulate the Pathfinder queries for evaluation. First, we needed to develop reasonable approximations to what a containing demonstration would look like, based on the existing CALO instrumentation. This lets us identify alternative sources for each target, based on what other actions would already have been executed and what were likely procedure inputs. This was primarily a thought exercise, based on similar CALO procedures that had previously been taught through demonstration and the use cases obtained from the ethnographic study. Figure 5 presents snippets from a demonstration for sending travel notification email to a traveler's supervisor and a corresponding approximation of a demonstration for sending a weekly report by email to the principal investigator of a project.

Pathfinder Query: *Person (traveler) name to (supervisor) EmailAddress*
Procedure Inputs: `+traveler_name="Julie Wong"`
Demonstration:
…
`(sendEmail +to="gondek@esd.sri.com" +subject= "Out on travel" …)`
…

Pathfinder Query: *Project Name to (PI) EmailAddress*
Procedure Inputs: *`+project_name="PExA"`*
Demonstration:
…
`(sendEmail +to="myers@ai.sri.com" +subject= "Weekly Report" …)`
…

Figure 5. Actual demonstration for sending travel notification via email and mapping to a corresponding demonstration for sending a weekly report via email.

Second, we needed to instantiate each use case for each CALO KB, since the use cases we identified in the user study were naturally specific to the user involved. To find these instantiations, we used various CALO tools for browsing the KBs, including the Query Manager interface for querying the KBs and the IRIS Data App for directly browsing certain classes of objects such as Persons and Documents. For example, in the case of sending a weekly report to a project PI, we ran a query to find projects that the CALO user was involved in and the leader for each of those projects. This provided us with a set of possible instantiations for simulating the case of the user sending a report to the PI of a project.

We had originally identified 14 use cases that could potentially be answered by Pathfinder based on what was captured by the CALO Ontology. However, we eliminated a few of these because of the difficulty of finding appropriate instantiations given lack of direct access to certain applications. For example, while all email messages are stored in the CALO KB, there is no easy way to browse them in a manner similar to looking at them through a mail application. So finding a message that would approximate the content required by a use case (e.g., a request for a report) would have been very difficult. We also discovered that although the ontology was capable of representing certain types of knowledge, this information was not always captured, whether by design or because of some error in the harvesting code. For example, the CALO KBs did not record the meeting organizers and the concept of Organization was improperly specialized to Project. The final set of 6 use cases used in the evaluation is summarized in Table 1.

EXPERIMENTAL EVALUATION

For each of the six CALO KBs chosen for evaluation, we arbitrarily selected two source-target pairs for each of the six use cases in Table 1. Use cases 5 and 6 define unique targets for a given user, but use cases 1 through 4 can each be instantiated in a number of ways for a CALO KB. For example, the target for use case 1 could be the principal investigator (PI) for any project the CALO user was involved in, while the target for use case 4 could be any participant of a meeting the CALO user also attended. When possible we chose targets covering not only entities in the Y4 Test but also other SRI and non-SRI entities. We also chose a few common targets across some of the CALO KB users (e.g., a meeting that both users attended). For each KB, we created two instantiations each of use cases 1 through 4 and used the unique instantiations of use cases 5 and 6.

Table 1. Use cases for Pathfinder evaluation.

Scenario	Use Case	Source	Target
Weekly Progress Report	1	project name	project PI (Person) email address
Quarterly Report	2	project name	EventEntry
Quarterly Report	3	project name	project participant (Person) full name
Meeting Reminder	4	EventEntry	meeting attendee (Person) email address
Vacation Notification	5	user (Person) name	supervisor (Person) email address
Vacation Notification	6	user (Person) name	admin (Person) email address

Table 2. Summary of Pathfinder results.

CALO KB	*Use Case*									
	1		*3*		*4*		*5*		*6*	
	#paths	*length*	*#paths*	*length*	*#paths*	*length*	*#paths*	*length*	*#paths*	*length*
A	a	**9**	6	**7** [3x]	2	**7**	4	**9** [4x]	8	**9** [4x]
	1	**9**	6	**7** [3x]	2	**7**				
B	4	9 (7)	6	**7** [3x]	3	**7**	8	**9** [4x]	8	**9** [4x]
	1	**9**	3		3	**7**				
C	5	9 (7)	12	**7** [3x]	3	**7**	8	**9** [4x]	8	**9** [4x]
	5	9 (7)	24	**7** [3x]	4	**7**				
D	3	9 (7)	24	**7** [3x]	3	**7**	8	**9** [4x]	8	**9** [4x]
	1	**9**	12	7 (5)	2	**7**				
E	1	**9**	3	**7** [3x]	3	**7**	8	**9** [4x]	8	**9** [4x]
	1	**9**	3	**7** [3x]	3	**7**				
F	1	**9**	6	**7** [3x]	3	**7**	8	**9** [4x]	8	**9** [4x]
	159	9 (5)	48	**7** [3x]	3	**7**				

n *correct path = shortest path*
n[ox] *correct path = one of o shortest paths*
m(n) *correct path of length m longer than shortest path of length n*

Use Cases Improved by Filtering Heuristics

prefer paths with fewer targets
remove redundant paths
remove repeated subpaths, remove longcuts

We ran Pathfinder on each source-target pair with a maximum search depth of 7 (maximum path length of 15) using restricted bidirectional search. For each pair, we tallied the alternative paths returned. Table 2 summarizes the results.

For each source-target pair, we list the number of paths returned together with the length of the correct path. In the case where the correct path is *not* the shortest path, we also indicate in parentheses the length of the shortest path (e.g., 9 (7)). To help visualize the results, we display in bold (e.g., **7**) those paths for which the default shortest path heuristic would have returned the right answer. In the case where the shortest path is correct but is also nonunique (i.e., there are other paths that are equally short), we indicate in square brackets the total number of shortest paths (e.g., [4x]). Because of an apparent deficiency in the relations considered by Pathfinder, it was unable to return the correct path for any of the instantiations of use case 2 and so we omit those results. Also, for KB B, Pathfinder was unable to return the correct answer for the second instantiation of use case 3.

These results show that the shortest path heuristic results in finding the correct path for use cases 4, 5, and 6. However, for use cases 5 and 6, the correct path is not a unique shortest path. The shortest path heuristic also succeeds in identifying the correct path for some, but not all, instantiations of use cases 1 and 3. We now analyze our proposed filtering techniques with respect to these results.

Prefer Paths with Fewer Targets

Recall that while a path connects a given source to its target, that path may connect the source to other targets as well. Thus, one heuristic we proposed was to prefer paths that led to fewer (ideally, just one) target. The value of this heuristic is illustrated in particular instantiations of use case 1 (project name to PI email address) for KBs B, C, and D. In all these cases, the correct path of length 9 had a single target while the shortest path had 7. The heuristic would also have helped in the second instantiation of use case 3 for KB D, where the correct path of length 7 had four targets, while the shortest path of length 5 had five. However, at least for the use cases in our evaluation, this heuristic often seems to be subsumed by the shortest path heuristic. That is, correct paths leading to fewer targets were also often the shortest paths. And in the case where the correct path was not the shortest path, it usually resulted in at least as many targets as the shortest path.

Remove Redundant Paths

Across all the KBs, for use cases 5 and 6 the correct path was only one of four shortest paths. This turned out to be due to the user's full name corresponding to three other properties. Specifically, for a CALO KB user, *personFullNameIs*, *personPreferredFullNameIs*, *caloIdIs*, and *displayNameIs* have the same string value—the person's full name. (An exception was KB A, for which *caloIdIs* was not set.) For the purpose of identifying the CALO user, these relations are redundant and the paths they yield are effectively interchangeable. By retaining only one from each set of equivalent paths, we end up with only two total paths, and just one shortest path (the correct path). The same issue caused the multiple shortest paths in use case 3, and removing the redundant paths would have been especially helpful in those instances where Pathfinder returned many (12, 24, 48) alternative paths.

Remove Repeated Subpaths, Longcuts

A particularly illuminating instance of the problem of repeated subpaths and longcuts is the second instantiation of use case 1 for KB F. Pathfinder returned 159 different paths between the source (the project name "OPI") and the target (the PI email address). The vast majority of the paths occurred because of the CALO KB user had three email threads that happened to start with a message having the subject line "OPI"—one thread of which included the PI as a recipient and another of which was marked as related to the project "OPI" (which is related to the PI). For either of these threads, there were essentially paths going through successively more messages in the thread before branching out to the PI. This was compounded by the existence of the *next-element* relation, which was redundant with *nextEmailMessageInThreadOf*.

Because of the repeatable nature of *nextEmailMessageInThreadOf*—that is, it can occur consecutively any number of times—redundant relations result in an exponential blowup in the number of paths. Given m redundant, repeatable relations and n repetitions, there are m^n equivalent representations of the same path. In this case, if we just remove all paths containing the redundant *next-element* relation, the number of candidate paths is reduced from 159 to 22.

The repeated *nextEmailMessageInThreadOf* relation presents another opportunity for further filtering. As discussed previously, in the case where the outgoing relation from each member of a repeating subpath is essentially the same, we can just keep the shortest member—the one where the relation occurs just once. Applying this filtering technique to the remaining 21 candidates further reduces the number of paths to nine.

Finally, in this particular case, even a single occurrence of *nextEmailMessageInThreadOf* is actually a longcut, since one can follow the same relation out to the target from the original message in the first place. So, finally, applying the filter for longcuts, we are left with just three paths, one of which is the intended path.

These filtering techniques overlap somewhat, since they all essentially remove paths that are in some way redundant with other paths. In this particular case, we could have reached the same final three paths simply by removing longcuts, since the $\{nextEmailMessageInThreadOf \mid next\text{-}element\}^*$ subpaths were all basically longcuts. As seen in the previous section, however, redundant relations can occur independently of repeated subpaths. It will also not always be the case that a repeated subpath is also a longcut and can be completely removed, as is the case with the responses in an email thread. Thus, all three filtering techniques are likely to be useful.

Prefer Common Paths

The final heuristic is that of preferring common paths when the target to be supported is part of a collection. This is manifested in use cases 3 and 4. For example, in use case 4 (meeting name to meeting attendee email address), there are actually n targets (email addresses) all contained in the

To field of the email message reminder. While we set out to test this heuristic by choosing different targets for the same source (i.e., different members for the same project, different attendees for the same meeting), because the correct path was already the shortest path, there was no real opportunity to test this heuristic. This filtering technique will be most useful in those cases where the elements of the target collection are each related to the source in a diverse set of ways. The KBs we used in the evaluation are perhaps not ideal for supporting such cases because of the somewhat controlled environment (i.e., the annual evaluation) in which they were developed.

DISCUSSION AND FUTURE WORK

Our experimental results provide evidence that filtering heuristics can be successfully used to narrow down the set of possible paths returned by Pathfinder. Recall that one of the primary motivations for doing so was to open up the possibility of interactive task learning—specifically, asking the user to select between alternative hypotheses during learning. While this interaction does impose an additional burden on the user, we believe that the better generalizations it will support—particularly when learning from a single example—will more than compensate provided we can make the interaction as simple and straightforward as possible. As we have discovered in some more recent deployments of LAPDOG, the procedures users wish to automate can sometimes be quite lengthy—taking over half an hour to demonstrate and yielding procedures hundreds of actions long. This makes it unlikely that users will be willing to provide more than one demonstration, making it even more important to learn the correct procedure from a single example

There do remain some outstanding issues, however. While our filtering heuristics address the problem of too many paths, they do not address the problem of the *right* path yielding multiple targets from the same source. In our experiment, many paths yielded more than 30 different targets. While these were usually incorrect (and longer) paths for our use cases, one can imagine other situations where they would actually be the correct path. A possible approach to this problem is to ask the user during execution to choose from among the different values. This requires developing techniques for organizing and prioritizing the alternatives to manage the presentation to the user.

Another issue is that the restricted bidirectional search we currently use is incomplete. It cannot, for example, find the project leader to project participant relation depicted in Figure 1. A first attempt at extending Pathfinder to multidirectional paths involved simply adding in all the inverse relations. However, even with the use of the efficient *A(k)* index, we have seen performance degrade dramatically even for relatively shallow search depths. A possible solution is to involve the user earlier in the pathfinding process to help reduce the branching factor. Consider, for example, how this might impact the instantiation of use case 1 discussed previously that resulted in 159 alternative paths. If the user had been able to identify at the very beginning that "OPI" was a project name rather than an email subject, the vast majority of the paths would never have been generated. In addition to this interactive path filtering approach, we are continuing to explore other techniques for reducing the number of alternative paths presented to the user, including cost-sensitive path filtering and path clustering.

A big advantage of task learning within CALO is the integrated learning and reasoning framework it provides for the electronic desktop. This includes the centralized knowledge base over which Pathfinder operates to find relations between objects observed in a demonstration. As we deploy LAPDOG in new domains, a challenge will be to develop a corresponding approach for the information sources available in that domain. A possibility is to develop other methods of dataflow completion. In particular, LAPDOG already has the ability to perform dataflow completion through the generation of an information-producing plan. By developing both a general-purpose library of information-producing actions and techniques for developing domain-specific ones, we can continue to leverage LAPDOG's ability to learn from demonstrations involving unobservable actions.

SUMMARY AND CONCLUSIONS

We have presented an approach to learning from demonstrations involving unobservable mental actions that fills in missing dataflow by inferring relations between observed objects through the Pathfinder utility. Through use cases extracted from a contextual inquiry study on learning from demonstration, we tested Pathfinder's ability to find relational paths. We verified that a simple shortest-path heuristic is sometimes insufficient for identifying the correct answer. We developed a number of filtering techniques and showed how they can effectively reduce the number of alternative paths to a meaningful set that could potentially be presented to a user in an interactive approach to learning from demonstration.

ACKNOWLEDGMENTS

We would like to thank Guizhen Yang for designing and implementing the Pathfinder utility for finding relations between objects in the CALO KB. We would also like to thank Tom Lee and Steven Eker for helpful discussions on the problem of too many paths and the development of filtering heuristics.

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. FA8750-07-D-0185/0004. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA, or the Air Force Research Laboratory (AFRL).

REFERENCES

1. Allen, J., Chambers, N., Ferguson, G., Galescu, L., Jung, H., Swift, M., and Taysom, W. (2007). PLOW: a collaborative task learning agent. *Proc. AAAI 2007*.

2. Barker, K., Porter, B., and Clark, P. A library of generic concepts for composing knowledge bases. *Proc. K-CAP 2001*.

3. Beyer, H. and Holtzblatt, K. *Contextual Design: Defining Customer-Centered Systems*. Morgan Kaufmann, San Francisco, CA, 1998.

4. Blythe, J. Task learning by instruction in Tailor. *Proc IUI 2005*.

5. Blythe, J., Kapoor, D., Knoblock, C. A., and Lerman, K. Information integration for the masses. *J. UCS Special Issue on Wrapping Web Data Islands*, 2008.

6. Burstein, M., Laddaga, R., McDonald, D., Benyo, B., Roberston, P., Hussain, T., Brinn, M., and McDermott, D. POIROT—Integrated learning of Web service procedures. *Proc. AAAI 2008*.

7. CALO: Cognitive Agent that Learns and Organizes. http://caloproject.sri.com/.

8. Chaudhri, V. K., Cheyer, A., Giuli, R., Jarrold, B., Myers, K. L., and Niekrasz, J. A case study in engineering a knowledge base for a personal assistant. *Proc. Semantic Desktop and Social Semantic Collaboration Workshop*, 2006.

9. Cypher, A. (Ed.). *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, MA, 1993.

10. Eker, S., Lee, T. J., and Gervasio, M. Iteration learning by demonstration. *Proc. AAAI Spring Symposium on Agents that Learn from Human Teachers*, 2008.

11. Garland, A. and Lesh, N. Learning hierarchical task models by demonstration. *MERL Technical Report TR2003-1*, Mitsubishi Electric Research Laboratories, 2003.

12. Gervasio, M., Lee, T. J., and Eker, S. Learning email procedures for the desktop. *Proc. AAAI 2008 Workshop on Enhanced Messaging*, AAAI Press (2008).

13. Huffman, S. and Laird, J. Flexibly instructable agents, *Journal of AI Research*, 3, 1995.

14. Kaushik, R., Shenoy, P., Bohannon, P., and Gudes, E. Exploiting local similarity for indexing paths in graph-structured data. *Proc. ICDE 2002*, IEEE Computer Society (2002).

15. Lieberman, H. (Ed.). *Your Wish is My Command: Programming by Example*. Morgan Kaufmann, San Francisco, CA, 2001.

16. Little, G., Lau, T. A., Cypher, A., Lin, J., Haber, E. M., and Kandogan, E. Koala: capture, share, automate, personalize business processes on the Web. *Proc. CHI 2007*.

17. Morley, D. and Myers, K. The SPARK agent framework. *Proc. AAMAS 2004*.