# TutorialPlan: Automated Tutorial Generation from CAD Drawings

**Wei Li**
Autodesk Research
Toronto, Ontario, Canada
wei.li@autodesk.com

**Yuanlin Zhang**
Texas Tech University
Lubbock, Texas, USA
y.zhang@ttu.edu

**George Fitzmaurice**
Autodesk Research
Toronto, Ontario, Canada
george.fitzmaurice@autodesk.com

## Abstract

Authoring tutorials for complex software applications is a time consuming process. It also highly depends on the tutorial designer's skill level and experience. This paper introduces an approach which automatically generates software tutorials using the digital artifacts produced by the users of a software program. We model this process as an optimal planning problem using software produced artifacts, software specifications and the human-computer interaction Keystroke-Level Model (KLM). We present TutorialPlan, an automated tutorial generator, which creates step-by-step text and image instructions from CAD drawings and helps users learn AutoCAD, a complex design and drafting software. In our tutorial generator, the optimal planning problem is represented and solved using DLV, a general Answer Set Programming (ASP) system. DLV offers a natural representation of both the problem and the heuristics needed to solve it efficiently. A user study shows that the tutorials generated by our system are comparable to those generated by experienced AutoCAD users.

## 1 Introduction

Learning to use software applications is a challenging problem due to the tension between the ever increasing number of features of many modern applications, which can contain hundreds to thousands of features, and the unwillingness of users to spend time learning anything about a software program that is beyond their immediate task needs at hand [Carroll and McKendree, 1987].

The software learning problem has been investigated in a long line of research literature (see [Grossman *et al.*, 2009] and references therein). Many types of learning aids have been proposed, including online help, video assistance and interactive tutorials [Sukaviriya and Foley, 1990; Fernquist *et al.*, 2011; Li *et al.*, 2012]. A majority of the existing work is tutorial based. The tutorials need to be authored by tutorial designers who are often software experts. They first identify and design tasks (problems) for the tutorial and then work out the steps to complete them. The steps are recorded and presented as text, images, or videos. The finished tutorials are shared online or distributed to the learners.

There are some issues with this traditional tutorial authoring process. First, there is a constant shortage of tutorial design experts while features of the software change frequently. In a time when globalization prevails, many software applications such as AutoCAD have to offer versions in local languages and need corresponding tutorials. Therefore, it is very difficult and time consuming to produce up-to-date high quality tutorials. Second, the sample tasks in a tutorial are usually simplified and/or artificial depending on the designer's experience and time constraints. It is hard for users to find tutorials relevant to what they need to accomplish from the limited, existing tutorials.

We also note the growing community of users who share their design data online. For example, CAD drawings are abundant on the Internet. When a user wants to create a drawing she is not familiar with, it is relatively easy for her to find a similar drawing on the Internet. It would be a great help if a tutorial could be generated from a sample drawing. Unfortunately, existing tutorial systems are not able to leverage those online resources yet.

To address the challenges and make use of the opportunities, we propose to automatically generate tutorials from digital artifacts produced by users of a software application, the specification of software functionalities and cognitive models on human behavior and learning. By automation, we minimize the use of human experts and the effort to produce tutorials for different software versions (e.g., due to the use of different languages or new features). The algorithms together with the cognitive model will produce high quality tutorials (e.g., in terms of improving users' productivity). More importantly, we are able to address the challenge of the shortage of sample tasks relevant to the users' work needs thanks to the abundant artifacts of sample tasks available online.

We present a prototype system, TutorialPlan, to generate tutorials from a given CAD drawing to teach a user how to create the same drawing. A main problem in building the tutorial is to generate a sequence of commands to render the drawing. In TutorialPlan, we formulate the problem as an optimal planning problem [Hart *et al.*, 1968; Ghallab *et al.*, 2004] where the objective is to minimize the human's effort to accomplish the drawing based on a cognitive model the Keystroke-Level Model (KLM) [Card

*et al.*, 1980; John and Kieras, 1996]. An answer set programming system (ASP) - DLV [Gelfond and Kahl, 2012; Eiter *et al.*, 2003] is employed to solve the optimal planning problem. Once the sequence of commands is obtained, a tutorial composer will create an image based and human friendly tutorial from the commands. A preliminary evaluation shows very promising results. After compared with hand-designed tutorials, a group of experienced software users agree that automatically generated tutorials are useful and would recommend them to novice users.

## 2 Related Work

In this section, we will review the work related to tutorial generation which spans both the AI and HCI communities.

**Demo-based tutorial system.** Many recent tutorial systems are demo-based systems. They record the process of users working on a task and generate step-by-step instructions by compiling a software event log, UI state changes and data revisions [Grabler *et al.*, 2009; Fernquist *et al.*, 2011; Chi *et al.*, 2012; Laput *et al.*, 2012]. Tutorial designers are necessary for those systems.

**Intelligent tutoring system.** The most relevant work in intelligent tutoring systems is on course generation or course sequence. In the existing work, e.g., [Peachey and McCalla, 1986; Woo, 1991; Elbeh and Biundo-Stephan, 2012], course generation is usually formulated as a planning problem based on a course model, student model and pedagogical model. The more challenging work here is to build good student and pedagogical models themselves while it is assumed that the course model has a hierarchical structure. A major difference between this work and ours is that the overall course content is usually decided a priori while our content is from the users on the fly because most software users already have tasks in their mind and they may not want to spend time working on/learning pre-defined content [Carroll and Rosson, 1987].

**Automated help system.** Another related area is automated help systems which capture knowledge about software specifications and user interface. Cartoonist [Sukaviriya and Foley, 1990] uses pre and post conditions associated to interactive actions to search for a chain of events that can generate animation scripts. H3 [Moriyon *et al.*, 1994] applies a forward chaining inference engine on the specifications of the application's user interface to generate hypertext-based help. A task-oriented help system was presented in [Pangoli and Paternó, 1995]. It allows the help designer create task specifications with temporal constraints among sub-tasks and derives help content from these specifications. Similarly, a task specification editor was provided to the designer in [Contreras and Saiz, 1996; Garcia, 2000].

Another class of help support systems employs the planning techniques to avoid manual construction of complex task models for tutorial generation as the work above. For instance, HICCUPS, a help system for configuring statistical analysis [McKendree and Zaback, 1988], generates plans from users' goal statement, knowledge about the software and the users' recent interactions with the interface. It assumes a hierarchical structure of the software tasks, and its algorithm depends heavily on the competence of the software user. Another example is SmartAide [Ramachandran and Young, 2005] which generates context-sensitive task help based on the user's request for help, the current workspace context and the library of actions available to the system to construct an action sequence achieving the user's goal. However, SmartAide was only designed for a pre-defined set of general tasks.

The difference between this work and ours lies in the difference between software tutorial and software help systems. A software tutorial typically walks users step-by-step through a complex process, so that users can follow along with the tutorial and learn how to use a feature or perform a task [Selber *et al.*, 1997]. Early automated help systems only provide guidance for basic interactions, such as invoking commands or requiring input [Sukaviriya and Foley, 1990; Moriyon *et al.*, 1994]. Task-oriented help systems go beyond single commands [Pangoli and Paternó, 1995; Contreras and Saiz, 1996; Garcia, 2000]. But they need to construct task models and design specifications. Even some systems provided task editors. Those tasks are not flexible. They depend on the UI design models and have to be pre-defined by help designers.

**Other differences.** There are other major differences between our work and the previous work: 1) The previous work does not involve much human computer interaction modeling or optimality of a plan. Instead, we employ the Keystroke-Level model (KLM) to identify optimal plans. 2) The knowledge of our domain lacks the hierarchical structure. As a result, a natural choice for us is to use answer set programming while a hierarchical task network (HTN) planner [Ghallab *et al.*, 2004] is sufficient for the work above. 3) Very few empirical studies were conducted to evaluate the planning systems in the previous work. No user study was reported in almost all of the reviewed work. 4) Finally, the goal in the previous work is usually formulated by users while in our system, any software artifact produced by the online community can be used as a goal.

## 3 Problem Definition

Given a task and a set of well specified software commands, the design process of a task oriented software tutorial can be formulated as a *planning problem*: given the specification of actions and a goal, find a sequence of actions, called a *plan*, which can achieve the goal. The actions are the commands of the software, and the goal is the task.

For a complex piece of software, the same goal can often be achieved by multiple action sequences. For example, in a drawing software, to produce 10 parallel lines of the same length, we could use the LINE command 10 times and input coordinates for 20 points. But a more human friendly approach is to draw the first line and use the COPY and PASTE commands to easily create the rest of the lines. Given this consideration, it is not sufficient to model the tutorial generation as a planning problem because many plans obtainable from a planning problem formulation are not acceptable by human users. Some restrictions have to be added to the plans such that they are more appropriate for human users.

The GOMS (shorthand for Goal, Operator, Methods and Selection rules) model [Card *et al.*, 1986], a well known the-

oretical concept in human-computer interactions, offers criteria for a good plan. It provides an engineering model to make a priori quantitative predictions of human performance in a human-computer interaction design. It can address the following issues: lower-level perceptual motor issues, the complexity and efficiency of the interaction procedures, and how all these activities are performed together [John and Kieras, 1996]. The Keystroke-Level Model (KLM) is a simpler GOMS model and has matured sufficiently to be used in actual design [Card *et al.*, 1980; John and Kieras, 1996]. It provides an estimation of the execution time for a task, which fits our problem well. KLM associates a time to each action and mental preparation to finish a task. So, the execution time of a sequence of actions is the total time of the actions and necessary mental preparation.

In our case, every plan (a sequence of action) has an execution time under KLM. One criteria for a plan to be acceptable is for its execution time to be minimal. Now the problem of generating an acceptable plan for a given software task can be modeled by optimal-planning problems [Hart *et al.*, 1968].

A *planning instance* is $(S, I, G, A, f, c)$ where $S$ is a set whose elements are called *states*, $I$ a state, called an *initial state*, $G$ a state, called a *goal* state, $A$ a set whose members are called *actions*, $f$ a *transition function* mapping a state and an action to a state, and $c$ a function mapping an action to a number, called *cost*. A *plan* is a sequence of actions $(a_1, ..., a_n)$ such that $f(f(...f(f(I, a_1), a2)..., a_n)) = G$. The *cost* of a plan $(a_1, ..., a_n)$ is the sum of the cost of each its actions, i.e., $\Sigma_{i=1}^{i=n} c(a_i)$. An *optimal planning problem* is to find a plan, with minimal cost, for a planning instance.

The tutorial generation problem is an optimal planning problem where a state is a set of rendered objects, an action a command, the initial state empty, the goal state a set of rendered objects, the cost of an action defined by the KLM model, and the transition function defined by the effects of the commands. More details are in the next section.

## 4 Optimal Planning Using DLV

In this section, we will present details of the representation and solving of the tutorial generation problem using an ASP system – DLV.

Answer set programming originated from non-monotonic logic and logic programming. It is a logic programming paradigm based on the answer set semantics [Gelfond and Lifschitz, 1988], which particularly offers an elegant declarative semantics to the negation as failure operator in Prolog. An ASP program consists of *rules* of the form:

$$l_0 \ or \ ... \ or \ l_m \ \text{:-} \ l_m, ..., l_k, not \ l_{k+1}, ..., not \ l_n.$$

where each $l_i$ for $i \in [0..n]$ is a literal of some signature, i.e., expressions of the form $p(t)$ or $\neg p(t)$ where $p$ is a predicate and $t$ is a term, *not* is called *negation as failure* or *default negation* and *or* is called *epistemic disjunction*. A rule without body is called a *fact*, and a rule without head a *denial*. The rule is read as: if one believes $l_m, ..., l_k$ and there is no reason to believe $l_{k+1}, ..., l_n$, she must believe $l_0$, or ..... or $l_m$. The answer set semantics of a program $P$ assigns to $P$ a collection of answer sets, i.e., interpretations of the signature of $P$ corresponding to possible sets of beliefs (i.e., literals). These beliefs can be built by a rational reasoner by following the principles that the rules of $P$ must be satisfied and that one shall not believe anything one is not forced to believe.

DLV also allows *weak constraints* of the form:

$:\sim \ \mathtt{l_1}, \ldots, \mathtt{l_k}, \mathtt{not\ l_{k+1}} \ldots \mathtt{not\ l_n}.[\mathtt{C} : \mathtt{1}].$

where the rule is a denial and C is the cost (a number) associated with the rule. DLV finds the best answer sets that minimize the total cost of the violated weak constraints.

ASP offers an elegant solution to the planning problem [Lifschitz, 1999; Gelfond and Kahl, 2012]. Weak constraints of DLV offers the construct needed for optimal planning [Eiter *et al.*, 2003].

In the following we will use DLV to represent the problem of tutorial generation from AutoCAD drawings. AutoCAD[1] is a complex professional design software for producing accurate drawings. We choose AutoCAD also because it has been widely used by over 10 million users and has a large number of existing drawings.

To make the presentation concise and easy to understand, some literals serving only the purpose of satisfying the syntactical requirements of the DLV system are omitted. As a result, the rules in this paper might not be acceptable by DLV without adding the missed literals.

**Input drawing.** Given an AutoCAD drawing, its objects are represented as facts. A line, with a unique name Id, between points (x1, y2) and (x2, y2), is represented by

$$\mathtt{object}(\mathtt{Id}, \mathtt{cline}(\mathtt{start}(\mathtt{x1}, \mathtt{y1}), \mathtt{end}(\mathtt{x2}, \mathtt{y2}))).$$

A circle of unique name Id with center at (x, y) and radius of r is represented by

$$\mathtt{object}(\mathtt{Id}, \mathtt{ccircle}(\mathtt{center}(\mathtt{x}, \mathtt{y}), \mathtt{radius}(\mathtt{r}))).$$

**State/initial state/goal state.** A *state* is a set of fluents. A *fluent* is a property of some objects that may change from time to time. The most important one is rendered(Id) denoting that the object Id has been rendered. Other fluents include information on the execution of a command, e.g., on(line) denoting that currently the command to draw a line is on. Many fluents such as rendered(Id) follow the inertia law: when an Id is rendered at one moment, it is still rendered at the next moment if there is no reason to believe it is not rendered. For any inertial fluent F, we have a fact:

$$\mathtt{fluent}(\mathtt{inertial}, \mathtt{F}).$$

The inertia law is represented as

$$\begin{aligned}
\mathtt{holds}(\mathtt{F}, \mathtt{Next}) \ \text{:-} \ & \mathtt{next}(\mathtt{I}, \mathtt{Next}), \\
& \mathtt{fluent}(\mathtt{inertial}, \mathtt{F}), \\
& \mathtt{holds}(\mathtt{F}, \mathtt{I}), \\
& \mathtt{not} \ \neg\mathtt{holds}(\mathtt{F}, \mathtt{Next}). \\
\neg\mathtt{holds}(\mathtt{F}, \mathtt{Next}) \ \text{:-} \ & \mathtt{next}(\mathtt{I}, \mathtt{Next}), \\
& \mathtt{fluent}(\mathtt{inertial}, \mathtt{F}), \\
& \neg\mathtt{holds}(\mathtt{F}, \mathtt{I}), \\
& \mathtt{not} \ \mathtt{holds}(\mathtt{F}, \mathtt{Next}).
\end{aligned}$$

[1] http://usa.autodesk.com/autocad/

where $\neg$ is the classical negation and $\texttt{next}(\texttt{I}, \texttt{Next})$ denotes the next step of I is Next and $\texttt{holds}(\texttt{P}, \texttt{I})$ denotes property P holds at step I. For readers not familiar with ASP, the inertia rules give an excellent example to show the difference between $\neg$ and *not*. $\neg\texttt{holds}(\texttt{F}, \texttt{Next})$ (the head of the second rule above) means that we believe fluent F does not hold at the next moment while $\texttt{not holds}(\texttt{F}, \texttt{Next})$ (in the body of the second rule) denotes that we have no evidence, or we do not know, that the fluent F holds at the next moment. So, the second rule is understood as if F does not hold at the current moment and we have no evidence to show F will hold at the next moment, F is believed to keep its state of not holding at the next moment.

The *initial state* is empty, i.e., no object is rendered yet. The *goal state* is that $\texttt{rendered}(\texttt{Id})$ holds for any object Id of the input drawing. To represent fluents and effects of actions, we introduce abstract consecutive time steps ranging from 0 to a fixed given number $n$, called *plan length*. Plan length is a *parameter of the DLV program* for the tutorial generation problem. The time step for the initial state is 0. Intuitively, normally there is an action occurring at each step. The goal is that after a certain step, all objects will be rendered by the commands (actions) before this step. It is represented by the following DLV rules.

$$\texttt{holds}(\texttt{existUnrendered}, \texttt{I}) \texttt{:-} \\ \texttt{not holds}(\texttt{rendered}(\texttt{ID}), \texttt{I}).$$

which reads for any step I, if there exists an object Id which is not believed to be rendered, then there exists an unrendered object, i.e., $\texttt{holds}(\texttt{existUnrendered}, \texttt{I})$.

$$\texttt{goal}(\texttt{I}) \texttt{:-} \texttt{step}(\texttt{I}), \texttt{not holds}(\texttt{existUnrendered}, \texttt{I}).$$

which reads for any step I, the goal is achieved, i.e., $\texttt{goal}(\texttt{I})$, if there does not exist an unrendered object. The following two rules together mean that the goal *must* be achieved at step I.

$$\texttt{success} \texttt{:-} \texttt{goal}(\texttt{I}).$$

$$\texttt{:-not success}.$$

**Actions and transition function for commands.** An AutoCAD command consists of a set of subcommands. Each subcommand is taken as an action. Normally it is very hard to have an explicit representation of the transition function of a planning instance. In fact, a typical way is to specify the executable condition and effects of the actions. We use line drawing as an example. We have two classes of actions $\texttt{click}(\texttt{line})$ and $\texttt{input}(\texttt{x}, \texttt{y})$ (note that different parameter $(\texttt{x}, \texttt{y})$ means different input actions), which can be represented as facts

$$\texttt{action}(\texttt{click}(\texttt{line})).$$

$$\texttt{action}(\texttt{input}(\texttt{x}, \texttt{y})).$$

We use $\texttt{occurs}(\texttt{A}, \texttt{I})$ to denote the occurrence of an action A at step I. The effect of $\texttt{click}(\texttt{line})$ at I is that the line icon is clicked, denoted by $\texttt{holds}(\texttt{clicked}(\texttt{line}), \texttt{I})$:

$$\texttt{holds}(\texttt{clicked}(\texttt{line}), \texttt{I}+1) \texttt{:-occurs}(\texttt{click}(\texttt{line}), \texttt{I}).$$

At any step I, the executable condition of $\texttt{click}(\texttt{line})$ is that it is not clicked yet:

$$\neg\texttt{occurs}(\texttt{click}(\texttt{line}), \texttt{I}) \texttt{:-holds}(\texttt{clicked}(\texttt{line}), \texttt{I}).$$

Line command is a pretty complex command. For example, after three distinct points are drawn, one can type letter C (hotkey for close subcommand), which automatically generates a line connecting the last and the first input point. DLV rules provide a convenient way to express this complex situation.

**Optimal plan generation.** Our solution was guided by a simplified KLM. The cost of each action can be represented by facts. While we generate a fixed-cost to estimate the average time to click the icon, each command icon has an individual cost estimate based on the command icon size and location in the interface. For example, if the action of pointing to a target with a mouse uses twice as much time as typing random letters. $\texttt{click}(\texttt{line})$ can be assigned to a cost of 2*t while $\texttt{input}(\texttt{x}, \texttt{y})$ to a cost of $n * t$ where $n$ is the length of input string, which can be represented as the facts:

$$\texttt{cost}(\texttt{click}(\texttt{line}), \texttt{2t}).$$

$$\texttt{cost}(\texttt{input}(\texttt{x}, \texttt{y}), \texttt{nt}).$$

To generate a plan to achieve the goal state, we need a rule to say that any action can occur at any step, in DLV:

$$\texttt{occurs}(\texttt{A}, \texttt{I}) \texttt{ or } \neg\texttt{occurs}(\texttt{A}, \texttt{I}) \texttt{:-step}(\texttt{I}), \texttt{action}(\texttt{A}).$$

Since in a tutorial, parallel actions are not desirable, we need a denial to express the constraint that no two actions are allowed at the same step:

$$\texttt{:-step}(\texttt{I}), \texttt{occurs}(\texttt{A1}, \texttt{I}), \texttt{occurs}(\texttt{A2}, \texttt{I}), \texttt{A1} != \texttt{A2}.$$

To make the plan optimal, i.e., the total cost of the actions in the plan should be minimal, we need the following weak constraint:

$$:\sim \texttt{occurs}(\texttt{A}, \texttt{I}), \texttt{action}(\texttt{A}), \texttt{cost}(\texttt{A}, \texttt{Cost}).[\texttt{Cost} : \texttt{1}].$$

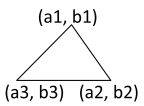
where $[\texttt{Cost} : \texttt{1}]$ associates the cost Cost of action A to the rule. Intuitively, this rule reads for any action A and step I, A is not allowed to occur at I and if the rule has to be violated, the total cost of the violated rules should be minimized.

Finally, we have the following result.

**Theorem 1** *Given a tutorial generation problem and* $\Pi$ *be a DLV program* $\Pi$ *with parameter* $n$ *for the problem, there is an optimal plan with finite number* $n$ *of actions iff there is an answer set of* $\Pi$.

**Example.** Given the input drawing of a triangle, line drawing command and the parameter plan length of 5, the following literals will appear in an answer set of the DLV program:

$\texttt{occurs}(\texttt{click}(\texttt{line}), 0)$,
$\texttt{occurs}(\texttt{input}(\texttt{a1}, \texttt{b1}), 1)$,
$\texttt{occurs}(\texttt{input}(\texttt{a2}, \texttt{b2}), 2)$,
$\texttt{occurs}(\texttt{input}(\texttt{a3}, \texttt{b3}), 3)$,
$\texttt{occurs}(\texttt{close}, 4)$




from which an optimal plan can be extracted: first click the

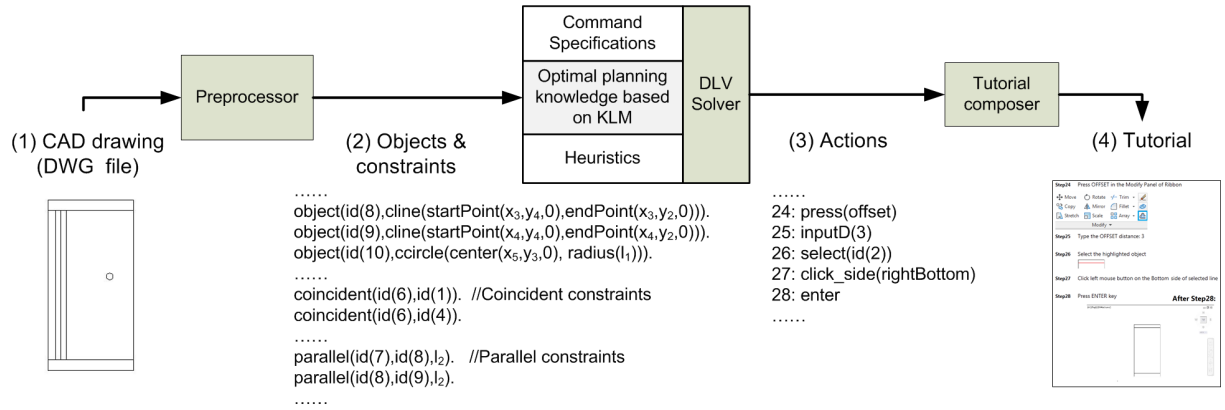



Figure 1: TutorialPlan system

line icon, then input the points $(a1, b1)$, $(a2, b2)$, and $(a3, b3)$ in sequence, and finally press letter C to complete.

**Heuristic knowledge.** Usually, efficiency becomes a problem for automated planning when the number of actions and steps increases. It is well known that extra knowledge can be used to improve the efficiency, and ASP offers the convenience in representing and reasoning with such knowledge. We give two such examples here. The first is the *non-overlapping* requirement which prohibits the overlapping of the commands drawing lines and circles. For example, the plan generation above allows such a overlapping plan "click line icon, click circle icon, input(a1, b1), inputCircle(a2, b2, R), ...." As a result, the search space for an optimal plan is huge (any action is allowed at any step). To prune the search space, we introduce a fluent $\texttt{on}(\texttt{Com})$ which denotes that command `Com` is on. The non-overlapping rule is:

$$\neg\texttt{occurs}(\texttt{click}(\texttt{C1})) \texttt{:-} \texttt{holds}(\texttt{on}(\texttt{C2}), \texttt{I}), \texttt{C1} \neq \texttt{C2}.$$

which reads for any step `I`, when command `C2` is on, no other click command is allowed. The second is for symmetry breaking. As an example, consider a drawing including many disjoint circles. Rules are written to make sure the circles are rendered in one specific order (e.g., from top left most to the right bottom most). Without such rules, DLV solver will search all different orders to render the same set of circles to make sure only plans with the lowest cost can be found.

## 5 TutorialPlan

Our system consists of three components: a preprocessor, a DLV solver based planner and a tutorial composer. The preprocessor automatically retrieves geometry objects and constraints from the AutoCAD drawing and translates them into facts. DLV solver starts the plan generation using these facts together with the software command specifications and knowledge on plan generation based on KLM. As the solution of an optimal planning problem, we get a series of actions, which contains information on commands and their parameters, together with a rendered object ID at each step. The tutorial composer then translates actions into steps where text and images are properly organized and compiled (see Figure 1 for an example).

Our current system modeled four AutoCAD command categories: Line tools, Circle tools, Offset and Join. Due to the requirement of accuracy and efficiency, a single AutoCAD command often contains multiple states, context-based parameters and complex workflows [Li *et al.*, 2012]. For example, it takes 5 steps to use the command offset to draw a parallel line which has a fixed distance to a selected line. There are 4 options to control the duplicated line, and 3 different workflows for doing multiple offsets.

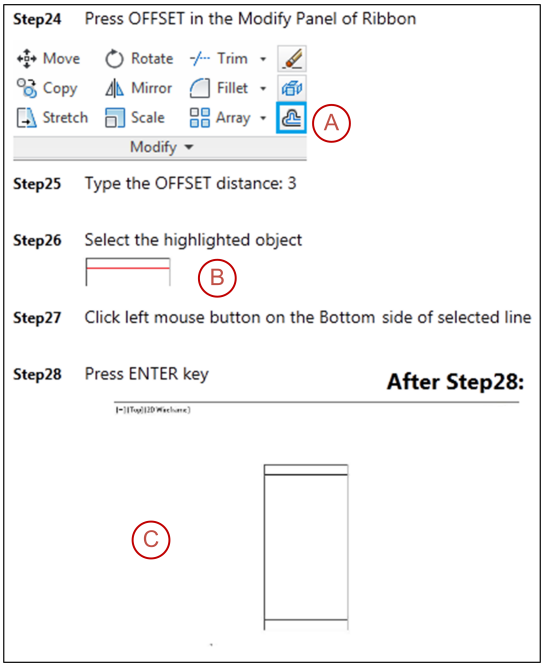



Figure 2: Generated tutorial at Figure 1(4)

Tutorial composer generates text and screen shots to help novice users follow the steps. Figure 2 shows a generated tutorial of drawing a door using line and offset. To help learners lunch the offset command, a screen captured image with the

highlighted offset button is included at step 24 (Figure 2(A)). Similarly, existing objects are highlighted to make the selection step easier (Figure 2(B)). At the bottom, an image of the latest drawing is automatically generated to show that a line will be created after the last step of the offset command (Figure 2(C)).

In order to further evaluate TutorialPlan, we tested it using an AutoCAD drawing benchmark library which contains 25 drawings. The drawings are either recreated using existing AutoCAD tutorials or copied from real world CAD drawings. Figure 3 shows six drawings from this library. Most of the drawings were solved by TutorialPlan in a reasonable time ranging from seconds to several hours (see Table 1).

When reading the time data, one needs to be careful that there is no simple characterization of the complexity of a drawing and the time to generate an optimal plan for it. For example, it seems that there are more actions needed for the drawing of E than those for F. However, we have only one type of objects, i.e., circles in E while we have several types of objects in F. To find an optimal way to draw a polyline (in F) is more complex than drawing circles that can interact with each other only in a very limited way. Note also that the circles in E are heavily constrained, which helps to reduce the search space significantly. Overall, it takes less time to solve the problem of F than that of E.

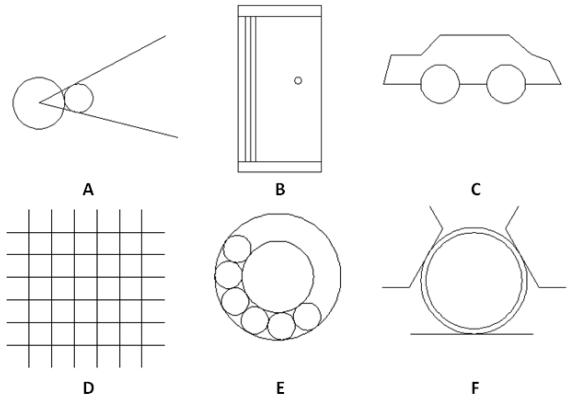



Figure 3: Examples from the benchmark library

| Example | #Objects | #Actions | Time (minutes) |
|---|---|---|---|
| A | 10 | 12 | 0.5 |
| B | 4 | 41 | 1 |
| C | 12 | 20 | 68 |
| D | 12 | 30 | 44 |
| E | 8 | 31 | 11 |
| F | 9 | 21 | 297 |

Table 1: Runtime of generating optimal plans for examples in Figure 3

## 6 User Study

We conducted a preliminary user study to compare our automatically generated tutorials with hand-designed tutorials. We hypothesized that TutorialPlan would generate similar step-by-step instructions as experienced AutoCAD users. We recruited 6 experienced AutoCAD users including three architects, one mechanical designer, and two CAD software testers (1 woman, 5 men). All participants used AutoCAD for more than 5 years on a regular basis, and half of them have over 10 years of experience. Each participant was asked to design two tutorials for novice users: Drawing A which contains 4 objects (Figure 3 (A)) and Drawing B which contains 10 objects (Figure 3 (B)). There were coincident, tangent and parallel constraints in those drawings. After all the tutorials were completed, we ask each participant to review tutorials designed by the other participants and the two tutorials generated by TutorialPlan generated. Thus, each person reviewed a total of 12 tutorials, 6 for each task. Participants don't know any authoring information related to each hand-designed or automatically generated tutorial.

The average time for authoring a tutorial was 30 minutes (minimum 10 and maximum 60 minutes) for Drawing A, and 33 minutes (minimum 15 and maximum 60 minutes) for drawing B. There were 6.8 steps on average (minimum 4 and maximum 14 steps) in hand-designed tutorials for A, and 9.8 steps on average (minimum 5 and maximum 20 steps) for B. The TutorialPlan generated tutorials consisted of 11 steps for A and 41 steps for B. Four participants used images and text in their tutorials, and two participants used only text. We followed each step in the hand-designed tutorials, and computed their expected performance time using KLM. Figure 4 shows the average expected task completion time for drawing A and B. We found that on average hand-designed tutorials are 25-34% slower than our tutorials. During the post

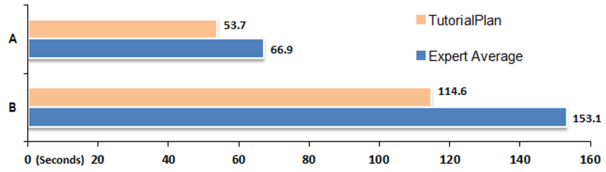



Figure 4: Average expected completion time based on KLM

study survey, each participant was asked to give scores to both the hand-designed and automatically generated tutorials. We measured four attributes of a tutorial: easiness to follow, usefulness, difficulty to author and willingness to recommend. Scores are from 1 (not support) to 6 (strong support). The survey responses are shown in Table 2. TutorialPLan tutorials top 3 out of 7 for all four questions. We found three main

| | Questions | Score for A | Score for B |
|---|---|---|---|
| 1 | Easy to follow | 3.7 (#3) | 4.2 (#1) |
| 2 | Useful tutorial | 5 (#1) | 5 (#1) |
| 3 | Difficult to generate | 5.5 (#1) | 5.8 (#1) |
| 4 | Would recommend to novice users | 3.8 (#2) | 3.5 (#2) |

Table 2: Generated tutorials' average scores and score rankings, in parenthesis, among 7 tutorials for drawing A and B

results. First, experienced AutoCAD users think that the tuto-

rials generated by TutorialPlan are useful and would recommend them to novice users. Second, participants think that all tutorials are difficult to generate. Third, TutorialPlan tutorials contain more detailed steps than hand-designed tutorials. It is a useful feature for novice users, but one participant commented that too much detail may make experienced users feel repetitive in some occasions. Solving the optimal planning problem makes the workflows in the generated tutorials have less expected completion times than hand-designed tutorials. But their workflows are similar to several hand-designed tutorials. We also found that the generated tutorials used a special *multiple* option in the offset command, which were not used in any hand-designed tutorials. This option did reduce a user's task completion time. Both TutorialPlan and expert tutorials obtaining high easy-to-follow scores have screen shot images for tracking drawing progress.

## 7 Conclusion, Discussion and Future Work

In this paper, we propose to generate tutorials automatically based on using the KLM model, software specifications and the abundant digital artifacts produced by users of software programs. This approach allows users to find interesting and relevant digital artifacts and have the system generate a tutorial to answer the question: How was that built? This serves as targeted learning. We have developed a prototype system TutorialPlan to compose tutorials automatically from a given CAD drawing, based on the commands generated from solving an optimal planning problem using an ASP program. We are able to generate tutorials from a set of sample CAD drawings, which were examples from existing AutoCAD tutorials. A group of experienced AutoCAD users evaluated the automatically generated tutorials with hand-designed tutorials. Our initial results show that the automatically generated tutorials were assessed to be similarly useful and of high quality compared to hand-designed tutorials.

In tutorial generation and knowledge representation, we believe it is novel to apply answer set programming with weak constraints (i.e., DLV) to a real life optimal planning problem in software learning. The problem has the following features. It is reasonably large in terms of plan length (up to 50 actions needed), number of commands (4 classes of commands: line, circle, join, and offset), and program size (2000+ lines of code). We observe the advantage of the ASP language in the following ways. First, we need to describe a dynamic domain with complex actions (e.g., the effects of the input point action) and relations among geometric objects (note that the connectedness of line segments and relations between compound objects formed by the join command are not covered in this paper due to the focus of this paper on the overall methodology). Declarativeness and non-monotonicity of ASP offers a natural modeling of this knowledge. Second, any interesting problem instances involve a large number of actions (partially because actions such as an input point are parameterized by all possible points in a drawing) and needs tens of steps to be accomplished. Aggravated by the optimality requirement, a minimal ASP program can easily run for days for just a very simple drawing even with the latest DLV solver. Heuristics have been very effective in boosting the performance of our program. Heuristics include no overlapping of commands of different nature (e.g., for lines vs. for circles), applying only relevant commands in a state, and breaking symmetry. ASP provides effective support on both representing and reasoning with the heuristic knowledge.

The initial success of our prototype system encourages us to continue the work along the following directions. 1) Extend our prototype system to include more typical AutoCAD commands and produce tutorials in other formats, such as video or interactive tutorials [Sukaviriya and Foley, 1990]. 2) Conduct a thorough user study. 3) Generate adaptive tutorials based on the current learner's skill level or from subdrawings where the learner is interested. 4) Examine other state of the art tools in solving the optimal planning problem such as PDDL and HTN planning systems [Ghallab *et al.*, 2004] with two purposes. First, identify the most effective solution for the tutorial generation problem. Second, offer feedback to improve the existing tools and systems to address challenges posed by the tutorial generation problem: the large amount of knowledge and actions that distinguish it from the existing benchmarks [Gerevini *et al.*, 2006].

In summary, we believe our approach is promising and unique within the domain of software learning.

## Acknowledgments

Yuanlin Zhang's work was partially supported by NSF grant IIS-1018031.

## References

[Card *et al.*, 1980] S.K. Card, T.P. Moran, and A. Newell. The keystroke-level model for user performance time with interactive systems. *Communications of the ACM*, 23(7):396–410, 1980.

[Card *et al.*, 1986] S.K. Card, T.P. Moran, and A. Newell. *The psychology of human-computer interaction*. CRC, 1986.

[Carroll and McKendree, 1987] J.M. Carroll and J. McKendree. Interface design issues for advice-giving expert systems. *Communications of the ACM*, 30(1):14–32, 1987.

[Carroll and Rosson, 1987] John M. Carroll and Mary Beth Rosson. Paradox of the active user. In *Interfacing thought: cognitive aspects of human-computer interaction*, pages 80–111. MIT Press, 1987.

[Chi *et al.*, 2012] Pei-Yu Chi, Sally Ahn, Amanda Ren, Mira Dontcheva, Wilmot Li, and Björn Hartmann. Mixt: automatic generation of step-by-step mixed media tutorials. In *Proc. of ACM UIST*, pages 93–102, 2012.

[Contreras and Saiz, 1996] Javier Contreras and Francisco Saiz. A framework for the automatic generation of software tutoring. In *Proc. of CADUI'96*, pages 171–182, 1996.

[Eiter *et al.*, 2003] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. Answer set planning under action costs. *J. Artif. Intell. Res. (JAIR)*, 19:25–71, 2003.

[Elbeh and Biundo-Stephan, 2012] H.M.A. Elbeh and S. Biundo-Stephan. *A Personalized Emotional Intelligent Tutoring System Based on AI Planning*. PhD thesis, Ulm University, 2012.

[Fernquist *et al.*, 2011] Jennifer Fernquist, Tovi Grossman, and George Fitzmaurice. Sketch-sketch revolution: an engaging tutorial system for guided sketching and application learning. In *Proc. of ACM UIST*, pages 373–382, 2011.

[Garcia, 2000] Federico Garcia. Cactus: automated tutorial course generation for software applications. In *Proc. of IUI*, pages 113–120, 2000.

[Gelfond and Kahl, 2012] Michael Gelfond and Yulia Kahl. *Knowledge Representation, Reasoning, and the Design of Intelligent Agents*. Manuscript, 2012.

[Gelfond and Lifschitz, 1988] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proceedings of ICLP-88*, pages 1070–1080, 1988.

[Gerevini *et al.*, 2006] A. Gerevini, B. Bonet, and B. Givan. Fifth international planning competition. http://www.plg.inf.uc3m.es/icaps06/preprints/i06-ipc-allpapers.pdf, 2006.

[Ghallab *et al.*, 2004] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning : Theory and Practice*. Morgan Kaufmann, 2004.

[Grabler *et al.*, 2009] Floraine Grabler, Maneesh Agrawala, Wilmot Li, Mira Dontcheva, and Takeo Igarashi. Generating photo manipulation tutorials by demonstration. In *ACM SIGGRAPH*, pages 1–9, 2009.

[Grossman *et al.*, 2009] Tovi Grossman, George Fitzmaurice, and Ramtin Attar. A survey of software learnability: metrics, methodologies and guidelines. In *Proc. of ACM CHI*, pages 649–658, 2009.

[Hart *et al.*, 1968] P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, 1968.

[John and Kieras, 1996] Bonnie E. John and David E. Kieras. Using goms for user interface design and evaluation: Which technique? *ACM Trans. Comput.-Hum. Interact.*, 3(4):287–319, 1996.

[Laput *et al.*, 2012] Gierad Laput, Eytan Adar, Mira Dontcheva, and Wilmot Li. Tutorial-based interfaces for cloud-enabled applications. In *Proc. of ACM UIST*, pages 113–122, 2012.

[Li *et al.*, 2012] Wei Li, Tovi Grossman, and George Fitzmaurice. Gamicad: a gamified tutorial system for first time autocad users. In *Proc. of ACM UIST*, pages 103–112, 2012.

[Lifschitz, 1999] Vladimir Lifschitz. Answer set planning. In *ICLP*, pages 23–37, 1999.

[McKendree and Zaback, 1988] J. McKendree and J. Zaback. Planning for advising. In *Proc. of ACM CHI*, pages 179–184, 1988.

[Moriyon *et al.*, 1994] Roberto Moriyon, Pedro Szekely, and Robert Neches. Automatic generation of help from interface design models. In *Proc. of ACM CHI*, pages 225–231, 1994.

[Pangoli and Paternó, 1995] S. Pangoli and F. Paternó. Automatic generation of task-oriented help. In *Proc. of ACM UIST*, pages 181–187, 1995.

[Peachey and McCalla, 1986] D.R. Peachey and G.I. McCalla. Using planning techniques in intelligent tutoring systems. *International Journal of Man-Machine Studies*, 24(1):77–98, 1986.

[Ramachandran and Young, 2005] Ashwin Ramachandran and R. Michael Young. Providing intelligent help across applications in dynamic user and environment contexts. In *Proc. of IUI*, pages 269–271, 2005.

[Selber *et al.*, 1997] Stuart A. Selber, Johndan Johnson-Eilola, and Brad Mehlenbacher. *Online Support Systems: Tutorials, Documentation, and Help*. CRC Press, 1997.

[Sukaviriya and Foley, 1990] Piyawadee Sukaviriya and James D. Foley. Coupling a ui framework with automatic generation of context-sensitive animated help. In *Proc. of ACM SIGGRAPH*, pages 152–166, 1990.

[Woo, 1991] C.W. Woo. *Instructional planning in an intelligent tutoring system: combining global lesson plans with local discourse control*. PhD thesis, Illinois Institute of Technology, 1991.