Proceedings of the Twenty-Ninth AAAI Conference on Innovative Applications (IAAI-17))

A Logic Based Approach to Answering Questions about Alternatives in DIY Domains

Yi Wang, Joohyung Lee
Arizona State University,
Tempe, AZ, USA
{ywang485, joolee}@asu.edu

Doo Soon Kim
Bosch Research and Technology Center,
Palo Alto, CA, USA
DooSoon.Kim@us.bosch.com

Abstract

Many question answering systems have primarily focused on factoid questions. These systems require the answers to be explicitly stored in a knowledge base (KB) but due to this requirement, they fail to answer many questions for which the answers cannot be pre-formulated. This paper presents a question answering system which aims at answering non-factoid questions in the DIY domain using logic-based reasoning. Specifically, the system uses Answer Set Programming to derive an answer by combining various types of knowledge such as domain and commonsense knowledge. We showcase the system by answering one specific type of questions — questions about alternatives. The evaluation result shows that our logic-based reasoning together with the KB (constructed from texts using Information Extraction) significantly improves the user experience.

Introduction

Despite the popularity of DIY (Do-It-Yourself), DIY is not an easy task due to one major problem: DIYers may have many questions about a project, but in many cases, professionals who can assist the DIYers are unavailable. For example, one might wonder about hypothetical questions such as *"Can I use a table saw instead of a jigsaw in step2"* or *"Is it ok to put this DIY product outside?"* These questions are hard, especially for novice DIYers, because answering those questions requires them to possess extensive domain knowledge and experience. If a question answering (QA) system can answer those questions and explain the reason, it will greatly benefit the DIYers.

It is, however, challenging to develop a QA system for DIYers. First, most recent QA systems have focused on factoid questions, e.g., (Ferrucci et al. 2010; Waltinger et al. 2013). These factoid QA systems find an answer by retrieving from their KB an answer closely matching the question, and thus require all candidate answers to be explicitly stored in the KB. However, for many DIY questions (such as the above hypothetical questions), it is infeasible to enumerate all possible questions in advance.

Another challenge for answering the non-factoid questions is that the system should not only provide an answer but also explain the reason for the answer. For example, for a question *"Can I use a table saw instead of a jigsaw in step2?"*, just answering "yes" or "no" is not helpful. Rather, the system should deliver the related information, for example, by saying *"It is not recommended because step2 requires curve cutting, which is not supported by a table saw"*. This explanation capability is not considered in most factoid QA systems.

To address these challenges, we present a novel QA system which uses Answer Set Programming (ASP) (Lifschitz 2008) to answer non-factoid questions on DIY projects.[1] Our system currently addresses one particular type of non-factoid questions, questions about alternatives (e.g., "Can I use an alternative tool/material instead of the suggested one under a certain circumstance?") but the general methodology can also be applied to other types of non-factoid questions. Using ASP, our system encodes various types of knowledge and constraints (domain knowledge, commonsense knowledge, constraints from users, contexts and projects) and derives an answer satisfying the question, the background knowledge and the constraints. ASP is particularly useful for our task because our KB includes many rules about commonsense knowledge (e.g., "Normally, a power tool is dangerous to kids") which can be easily represented by ASP rules.

To evaluate our system, we performed a user study where we compared our system against online search, a common practice for information seeking. Our result shows that online search is unsuitable for answering non-factoid DIY questions; it takes too much time, and even many participants failed to find an answer. However, a majority of participants are satisfied with our system because it promptly presents an answer, and the explanation delivered by our system is informative. ASP has been previously applied to other QA systems (e.g., (Aditya et al. 2015)) to answer complex questions but, to our knowledge, our system is the first to apply ASP to non-factoid DIY questions in order to derive an answer under various types of DIY-related knowledge and constraints.

The paper is organized as follows. After the review of ASP, we give an overview of the whole framework, and then introduce the ASP rules and facts that are used to answer the

[1] https://www.bosch-do-it.com/za/en/diy/knowledge/project-guides/index.jsp

questions about alternatives. Then, we present our evaluation result. Finally, we discuss how to extend this framework to offer more advanced reasoning.

Background: Answer Set Programming

Answer Set Programming (ASP) is a declarative programming paradigm that is suitable for the design and implementation of knowledge-intensive applications. It has emerged from the interaction between two lines of research – nonmonotonic semantics of logic programs and application of satisfiability solvers to search problems. The idea is to represent a search problem by a logic program whose intended models, called "answer sets," correspond to the solutions of the problem, and then find these models using an answer set solver, such as CLINGO and DLV.

The language of ASP is logic programs under the stable model semantics (Gelfond and Lifschitz 1988), which allows for elegant representation of several aspects of knowledge such as causality, defaults, incomplete information, preference, and recursive definitions. What distinguishes ASP from other nonmonotonic formalisms is the availability of several efficient answer set solvers, which led to practical nonmonotonic reasoning that can be applied to industrial level applications.

In our framework, we apply answer set programming in two phases — information extraction to populate knowledge base and inference to derive an answer from the knowledge bases. The following are the brief introduction of some of the features of ASP that we found useful in our work. The rules are written in the language of CLINGO v3.0.5. We refer the reader to the CLINGO manual (http://potassco.sourceforge.net) for their precise meaning.

Generate and Test Paradigm In ASP, one can easily describe a set of "potential solutions" and use a set of constraints to eliminate all "bad" solutions. For example, the following program finds tools `X` to suggest by ruling out expensive tools and unsafe tools:

```
3 {suggestedTool(X) : tool(X)} 5.
:- expensive(X), suggestedTool(X).
:- unsafe(X), suggestedTool(X).
```

The first rule is a *cardinality* constraint to generate answer sets that contain 3 to 5 suggested tools. The next two rules, which have the empty head, eliminate the answer sets among them which contain expensive or unsafe tools.

Aggregates The availability of aggregates, such as `#count`, `#sum`, `#min`, `#max`, in ASP allows one for representing combinatorial choices succinctly. For example, the following rule asserts that "the project `PROJ` is considered as `drilling-intensive` if there are at least 5 `drilling` steps".

```
drilling_intensive(PROJ) :-
   5{step(PROJ,ID,"drilling"):step_ID(ID)}, project(PROJ).
```

The following rule is another example, which estimates the total time `T` needed for a project by summing up times `T1` needed by each action `ACT` in the project:

```
time_estimation(PROJ,T) :- T = #sum[step(PROJ,ID,ACT):
                  time_needed(ACT,T1)=T1], project(PROJ).
```

Recursive Definition Another convenient feature of ASP is being able to represent recursive definitions, such as transitive closure. For example, the following rules define the relation `descendent_step` as the transitive closure of the relation `substep`.

```
descendent_step(ST1,ST2) :- substep(ST1,ST2).
descendent_step(ST1,ST2) :-
   descendent_step(ST1,ST3), descendent_step(ST3,ST2).
```

System Overview

Figure 1 illustrates the overall QA system, where arrows indicate the information flow. The core part of the system is an ASP solver, which answers the questions based on the relevant information from the knowledge base and the user profile. The knowledge base contains a *project knowledge base* and a *domain knowledge base*. The domain knowledge base provides project-independent information, such as attributes of tools, accessories, and materials. The project knowledge base provides project-specific information, such as tools and materials needed and steps to be performed on a certain project. The user profile provides information on the user who is asking the question, such as his/her DIY skill level and age. Information on the specific project that is being performed, along with the user profile, facilitates context-aware question answering for the DIY domain. They are integrated with the domain knowledge by a set of project constraint generation rules. The domain knowledge base is constructed by automatically extracting information from web documents guided by an ontology manually created by woodworking experts. The question is asked in natural language, and a question interpreter converts the question into a structured form that can be part of the input to the ASP solver.

Answering Questions on Alternatives

In this section, we explain how the framework addresses a specific type of questions — questions about alternatives. These questions are about the possibility of using one object to replace another object in performing a certain project. An example of this type of questions is

> Can I use a table saw instead of a jigsaw in the project "armchair with stool"?

We call the object to be replaced "*target object*," and the object that replaces the target object "*new object*." The answers to this type of questions need to take into account the following factors:

1. The basic function of target objects and new objects, for example, "cutting" for saws, "sanding" for sanders.
2. Features that distinguish one type of objects from another type of objects under the same category. For example, both jigsaws and table saws are used for cutting, but jigsaws can make curved cuts while table saws generally cannot.
3. The role that the target object plays in the specific project. For example, a jigsaw can be used to cut curves in project A but can be used to cut straight lines in project B.

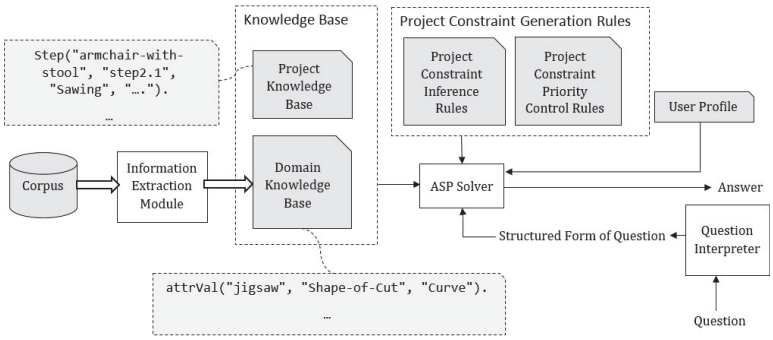

Figure 1: System Overview

4. Information about the user, such as the user's age, the user's skill level, etc.
5. Additional context information that may or may not be specified by the user, such as a time constraint for the project.

Among these aspects, the first and the second are provided by the domain knowledge base. The first is specified by woodworking experts. The second is automatically constructed from documents in natural language, guided by attribute schemas that are manually created by woodworking experts. The third is automatically inferred from project descriptions. The fourth and the fifth are specified by the user. All the information mentioned here is converted into ASP facts. With this information available, an ASP reasoner infers constraints on the new object, answers the question by checking whether these constraints are satisfied by the new object, and outputs additional information about this replacement that supports the answer.

Building the Knowledge Base

As mentioned before, among the five aspects that we take into account to answer questions about alternatives, the second one needs to be provided by the domain knowledge base. The Information Extraction Module (IE Module) is responsible for building this part of domain knowledge base. In this section, we discuss how we perform the information extraction. Figure 2 illustrates the whole IE process.

Relevant Sentence Collecting The first step is to create a collection of sentences that potentially contains the information we need. We collected about 2000 documents from various websites including tool buying guides, woodworking blogs, encyclopedia, etc. A dictionary of entities of interests is defined, including 16 types of saws, 42 types of wood material, 6 types of sanding tools, 10 types of fastening tools, 5 types of finishing tools and 5 types of measuring tools. We use this dictionary to select the sentences where at least one entity of interest appears from the corpus, and in this way form the collection of relevant sentences.

Proposition Extraction Extraction from the text is performed mainly via ASP reasoning. We run an ASP solver on an ASP program which contains rules to extract useful parts of a sentence to form an atomic proposition. For example, from the sentence "*Jigsaws are used to cut curves*" we extract the atomic proposition "`cut_curve(jigsaw)`."

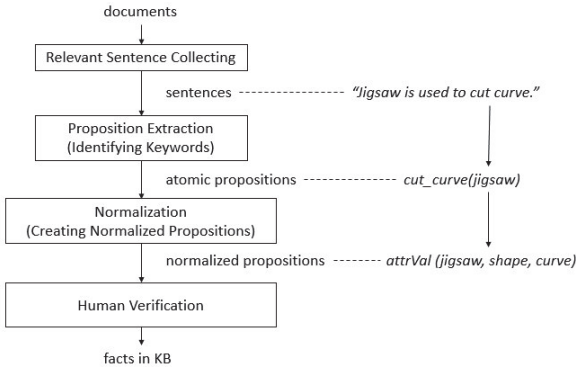

Figure 2: Knowledge Base Construction Pipeline

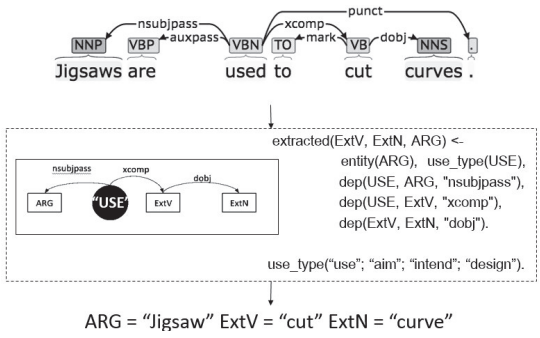

Figure 3: Proposition Extraction

Figure 3 illustrates the process. The input is the dependency parsing (De Marneffe et al. 2006) of the sentence, converted into a set of ASP facts. The ASP program for proposition extraction contains various sentence patterns defined as abstract dependency parsing, specifying which nodes in the dependency parsing tree should be extracted. For the above example, the sentence pattern (as shown in the middle box in Figure 3) is "ARG is/are used to $ExtV$ $ExtN$," where ARG, $ExtV$ and $ExtN$ are to be extracted.

Sometimes the modifiers of an extracted token contain

key information as well and thus need to be extracted. To do this, we first define what a modifier is:

```
modifying_relation("nmod").
modifying_relation("nmod:of").
...
modifies(MOD,TOK) :- dep(TOK,MOD,R), modifying_relation(R).
```

Then we write a rule saying "a modifier of an extracted token is also extracted."

```
extracted(MOD) :- extracted(TOK), modifies(MOD, TOK).
```

Note that this rule defines the concept "extracted" in terms of itself. Handling such recursive definition is a unique feature of ASP.

Normalization Having a set of keyword lists extracted from the text, the next step is to decide for each list whether the list contains any useful fact about the entity, and if yes, we derive a KB instance from it, or discard it otherwise. We do this by defining an attribute schema for each type of entities. An attribute schema contains a set of attributes along with their values. For example, we define that cutting tools have the attribute "`shape-of-cut`", whose values could be "`straight`", "`curve`", "`miter-cut`", "`bevel-cut`", etc. Note that in our schema, we allow an attribute to have multiple values or no values.

To extract attribute values from extracted propositions, we further define lexicons for each attribute value. For example, the lexicons for the attribute value `straight` of `shape-of-cut` are: "`straight`", "`rectangular`", "`stiff`", etc. We then simply detect if any of these lexicons appears in the extracted proposition. If yes, we construct an ASP fact of the form `attr(X, "Shape-of-Cut", "Straight")`.

Inferring attribute values via commonsense rules Since some attribute values are not specified in the texts, those values were inferred using ASP rules such as "If a tool is a power tool, it is difficult to use."

Attribute values inferred by these rules are used only when there is no value extracted from the text. This is implemented utilizing ASP's feature of default reasoning. Take the rule about power tool and difficulty of use as an example. We first write ASP rules to define "`Easy`" and "`Difficult`" as mutually exclusive values, and then we write:

```
% By default, power tools are difficult to use
{attr(X,"Difficulty","Difficult")} :- hasModifier(X, "power").
```

Inferring the Constraints

Having the domain knowledge base available, we can now write ASP rules to infer constraints that the new object needs to satisfy. Some examples of constraints inference rules are:

- **Constraints from project** if any lexicon for "curve" appears in the description of a step involving cutting, require the cutting tools used for the project to have the feature "Shape: Curve":

```
step_type(P, ST, "Curve_Cutting") :-
   step(P, ST, Act, Desc), step_type(P, ST, "Cutting"),
   @meansCurveCutting(Desc) == 1.
req(P, "Cutting", "Shape:Curve") :-
   step_type(P, ST, "Curve_Cutting"), project(P).
```

  In the above code, "`@meansCurveCutting(Desc)`" is an external Lua function call which returns 1 iff there is any lexicon defined for curve-cutting appearing in the string `Desc`.

- **Constraints from DIYer** "Kids-friendly" is defined as "easy to use and safe"; If the user's age is less than 18, require tools to be kids-friendly.

```
hasFeature(X, "Inferred:NotHardToUse") :-
   not attrVal(X, "Difficulty", "Difficult"), entity(X).
hasFeature(X,"Inferred:Kids-Friendly") :-
   attrVal(X,"Safety","Safe"),
   hasFeature(X,"Inferred:NotHardToUse"), entity(X).

req(P, X, "Inferred:Kids-Friendly") :-
   user_age(AGE), AGE < 18, action(X),
```

- **Constraints from user** If the user specified a time constraint which is smaller than the estimated project finishing time plus a time buffer, require tools to have high efficiency.

```
req(P,X,"Efficiency:High") :-
   user_time_budget(TH), total_time_needed(P,TM),
   query(P,T1,T2), TH * 60 < TM + TB * 60,
   time_buffer(TB), action(X).
```

Answering the Question

Having the domain knowledge base and the requirements on the new object, we are ready to answer the question about alternatives. We again use ASP rules to define what the answer to the question is.

We first define the relations between entities and actions. The ontology already specifies the corresponding actions of tools, such as the relation `hasAction`[2]. The ontology also specifies interchangeable actions, such as nailing and screwing[3]. We define the relation `hasDerivedAction` in terms of the relations `hasAction` and `interchangeable`. We say an entity `T` has derived action `Act` if `T` has action `Act`, or `T` has an action `Act1` that is interchangeable with `Act`

```
hasDerivedAction(T,Act) :- hasAction(T,Act).
hasDerivedAction(T,Act) :-
  hasAction(T,Act1),
  interchangeable(Act, Act1).
```

Then we define when the replacement is not recommended. The replacement is not recommended in the following two cases:

- The replacement of `T1` by `T2` is not recommended if `T2` does not support the action that `T1` has.

```
not_recommended(P,T1,T2) :- query(P,T1,T2),
   hasDerivedAction(T1,Act), not hasDerivedAction(T2,Act).
```

- The replacement of `T1` by `T2` is not recommended if `T2` supports all actions that the target entity has, but does not support some features that are required by the project

[2]For example, `hasAction("Screwdriver", "Screwing")`.

[3]We consider two actions as interchangeable actions if it is generally possible to replace one by the other. It does not mean it is always ok to replace one by the other.

```
not_recommended(P,T1,T2) :- query(P,T1,T2),
   req(P,Act,Req), hasDerivedAction(T2, Act),
   not hasFeature(T2, Req),
```

Otherwise, the replacement is possible.

```
possible(P,T1,T2) :- not not_recommended(P,T1,T2),
  query(P, T1, T2).
```

Result

The final system consists of $\sim$800 ASP facts about entity features (extracted from the text), $\sim$ 700 ASP facts converted from the ontology, $\sim$8800 ASP facts converted from 44 project instructions and $\sim$100 ASP rules.

Table 1 shows four question answering examples to illustrate the system's ability to perform context-aware question answering.

Evaluation

To evaluate our system, we compared our system against online search, a common practice for information seeking, on 41 questions about alternatives.[4] To do this, we recruited about 250 amateur DIYers through Amazon Mechanical Turk,[5] and for each question, we asked three participants to find an answer using online search and report the answer, the total time taken and the number of the webpages visited. If the time exceeded three minutes, we considered it to *fail*. For our system, we did not measure the total time taken since the answer was instantaneously found. Instead we asked the two questions: (1) whether the answer exhibited sufficient domain knowledge and (2) whether the answer would be helpful for their potential DIY projects.

Our result shows that our system provides a much better user experience. First, we find that 40% of the participants failed to find an answer within three minutes and that the average number of the websites visited by the participants was three. This indicates that answering a non-factoid DIY question using online search requires quite much time and efforts. Fig 5 shows the result of our system. The feedback from the participants is positive. 67% participants responded that the answer provided by our system showed sufficient domain knowledge, and 83% participants responded that the answers would be helpful for their future DIY projects.

Extension: Incorporating Quantitative Uncertainty in Answers

Our basic reasoning provides deterministic answers to questions about alternatives - an either-or "possible/recommended" answer and a comparison between the new object and the target object where a list of Boolean-valued features is shown. In a real-world setting, often it is more reasonable to give an answer with a certainty degree (e.g., "The replacement is **mildly/strongly** recommended", etc.). This

[4]These questions were collected through a user study before the development of the system. In the user study, we asked DIYers to perform a DIY task while speaking aloud any question and then recorded those questions.

[5]http://www.mturk.com

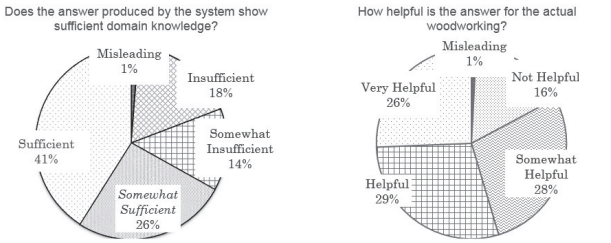

Figure 4: Results of the Survey

also applies to the comparison between two objects. Two objects might have the same feature to different degrees. It is desirable if in the comparison we can say things like "A is **way more** efficient than B," "A is **slightly less safe** than B," etc. to describe the situation more precisely. Here we briefly illustrate how this can be implemented in our framework using ASP.

First, we define each constraint as a weighted sum of a set of entity features (seen as 1-0 variables). We take "kids-friendly" and "high-efficiency" for example:

```
contribute("Difficulty:Easy","Kids-Friendly",10).
contribute("Difficulty:Difficult","Kids-Friendly",-10).
contribute("Safety:Safe","Kid-Friendly",20).
contribute("Safety:Unsafe","Kids-Friendly",-50).
contribute("Efficiency:High","High-Efficiency",50).
contribute("Efficiency:Low","High-Efficiency",-50).

constraint_sat(TAM, CONSTR, X) :-
   entity(TAM), constraint(CONSTR),
   X = #sum[hasFeature(TAM,F):contribute(F,CONSTR,Y) = Y].
```

Then we define the certainty degree of recommendation as the weighted sum of the degree to which all constraints are satisfied:

```
all_constraint_sat(TAM,X) :- entity(TAM),
   X = #sum[hasConstraint(CONSTR,W):
            constraint_sat(TAM,CONSTR,Y) = Y*W].
```

Finally, we define the advantage and disadvantage of the new object relative to the target object, as the difference in the degrees to which each of the two objects satisfies each constraint

```
pros(CONSTR,Z) :- query(P,TARGET_TAM,NEW_TAM),
   constraint_sat(NEW_TAM,CONSTR,X),
   constraint_sat(TARGET_TAM,CONSTR,Y), Z = X - Y, Z > 0.
```

As an example, suppose the user asks whether it is ok to use jigsaw instead of hand saw in some project:

```
query("some-project", "hand-saw", "jigsaw").
```

And the two constraints "kids-friendly" and "high-efficiency" are inferred for the project, with "kids-friendly" considered much more important than "high-efficiency":

```
hasConstraint("Kids-Friendly", 10).
hasConstraint("High-Efficiency", 1).
```

where 10 and 1 represent how important each of the constraints is. In real applications, these numbers can be derived from the context. For example, the importance of "high-efficiency" can be negatively correlated to the difference between time budget and estimated project finishing time; the

Question	Context	Answer
Can I use table saw instead of jigsaw?	Project: bird house for balcony and garden[1] (where jigsaw is used cut straight lines)	Possible. Please be aware of the following differences between the two entities: jigsaw: "Shape:Curve" "Shape:Circle" "Portability:Portable/Stationary" table-saw: "Shape:Ripcut" "Shape:Bevel-Cut" "Shape:Miter-Cut" "Portability:Stationary"
	Project: armchair with stool[2] (where jigsaw is used cut curves)	Not Recommended. table-saw does not support "Shape:Curve".
Can I use nail gun instead of screwdriver in the project armchair with stool?	User Age: Adult	Possible. Please be aware of the following differences between the two entities: screwdriver: "Strength:High" "Durability:High" "Cost:High" nail-gun: "Safety:Unsafe" "Strength:Medium" "Durability:Medium" "Cost:Medium"
	User Age: Kid	Not Recommended. nail-gun does not support "Kids-Friendly".
Can I hand sand the surface instead of using multi-sander in the project treasure chest?[3]	Time constraint: 5 hours	Possible. Please be aware of the following differences between the two entities: multi-sander: "Quality:Precise" "Kids-Friendly:No" "Surface-Size:Large" "Power-Source:Corded" "Material:Wood" "Shape:Straight" "Quality:Medium" "Portability:Portable/Stationary" "Safety:Unsafe" "Difficulty:Difficult" "Efficiency:High" hand sander: "Efficiency:Low" "Power-Source:Manual" "Portability:Portable" "Quality:Rough
	Time constraint: 3 hours	Not Recommended. hand sander does not support "Efficiency:High".

[1] https://www.bosch-do-it.com/za/en/diy/knowledge/project-guides/birdhouse-for-balcony-and-garden-68609.jsp
[2] https://www.bosch-do-it.com/za/en/diy/knowledge/project-guides/armchair-with-stool-68485.jsp
[3] https://www.bosch-do-it.com/za/en/diy/knowledge/project-guides/treasure-chest-83813.jsp

Table 1: Showcase Examples of Question Answering

importance of "kids-friendly" can be negatively correlated to the user's age.

The domain knowledge base provides:

```
entity("jigsaw").
hasFeature("jigsaw", "Difficulty:Difficult").
hasFeature("jigsaw", "Safety:Unsafe").
hasFeature("jigsaw", "Efficiency:High").
entity("hand-saw").
hasFeature("hand-saw", "Difficulty:Easy").
hasFeature("hand-saw", "Safety:Moderate").
hasFeature("hand-saw", "Efficiency:Low").
```

Combining all the above, the ASP solver returns

```
constraint_sat("hand-saw","High-Efficiency",-50)
constraint_sat("hand-saw","Kids-Friendly",10)
constraint_sat("jigsaw","High-Efficiency",50)
constraint_sat("jigsaw","Kids-Friendly",-60)
all_constraint_sat("hand-saw",50)
all_constraint_sat("jigsaw",-550)
recommendation_level("jigsaw",-550)
pros("High-Efficiency",100)   cons("Kids-Friendly",70)
```

which can be interpreted as "It is strongly not recommended to replace hand saw with jigsaw. Jigsaw is much more efficient than hand saw, but it is less kids-friendly than hand saw".

Conclusion

We presented a framework for non-factoid question answering in DIY domains. We took a logic-based approach to such non-factoid question answering. We showed how a specific type of questions — questions about alternatives was handled in this framework, from constructing the knowledge to the reasoning process involved in answering this type of questions. We also discussed how more advanced reasoning can be implemented in the framework. The evaluation result suggests our approach is promising. The future work is to extend the framework to account for other types of hypothetical reasoning in DIY domains.

Acknowledgments We are grateful to the anonymous referees for their useful comments on the draft of this paper. This work was partially supported by the National Science Foundation under Grants IIS-1319794 and IIS-1526301.

References

Aditya, S.; Baral, C.; Vo, N. H.; Lee, J.; et al. 2015. Recognizing social constructs from textual conversation. In *NAACL HLT 2015*, 1293–1298.

De Marneffe, M.-C.; MacCartney, B.; Manning, C. D.; et al. 2006. Generating typed dependency parses from phrase structure parses. In *Proceedings of LREC*, volume 6, 449–454.

Ferrucci, D.; Brown, E.; Chu-Carroll, J.; Fan, J.; Gondek, D.; Kalyanpur, A. A.; Lally, A.; et al. 2010. Building watson: An overview of the DeepQA project. *AI magazine* 31(3):59–79.

Gelfond, M., and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proceedings of International Logic Programming Conference and Symposium*, 1070–1080. MIT Press.

Lifschitz, V. 2008. What is answer set programming? In *Proceedings of the AAAI Conference on Artificial Intelligence*, 1594–1597. MIT Press.

Waltinger, U.; Tecuci, D.; Olteanu, M.; Mocanu, V.; and Sullivan, S. 2013. USI answers: Natural language question answering over (semi-) structured industry data. In *IAAI*.