Using AnsProlog with Link Grammar and WordNet for QA with deep reasoning

Luis Tari and Chitta Baral

Arizona State University
Department of Computer Science and Engineering
Brickyard Suite 501, 699 South Mill Avenue, Tempe, AZ 85287
{luis.tari, chitta}@asu.edu

Abstract

Early question and answering (QA) systems focused on keyword search among documents for answers. However, such systems can only answer fact-based questions. It becomes clear that to answer more sophisticated questions, QA systems should rely on some domain knowledge. We propose a question and answering system that uses AnsProlog to represent and reason from the knowledge extracted by using Link Grammar and WordNet.

Introduction

Consider the following simple scenario:

John flew from Paris to Baghdad. Is John in Baghdad?

If a QA system does not know the fact that a person *P* flying from place *A* to place *B* would imply that *P* is in *B*, then it is hard to answer correctly even for such a simple scenario. This simple scenario illustrates that domain knowledge is needed to answer certain questions. In other words, the answers may not be explicitly mentioned in the story, so it is necessary to incorporate domain knowledge to the existing facts from the story. We call such kind of questions and answering that require domain knowledge as QA with deep reasoning.

Several efforts have been made to incorporate certain world knowledge to QA systems so that non-fact-based or complex questions can be answered. In (Harabagiu 2001) a QA system was proposed to utilize various software agents that search and retrieve information to acquire knowledge that might be useful in answering complex questions. Semantic approaches to QA systems using WordNet were proposed in (Vicedo 2000, Pasca and Harabagiu 2001). Logic-based approach to QA systems were presented in (Harabagiu and Pasca and Maiorano 2000, Pasca 2000). Rus (Rus 2002) incorporated extended WordNet (Harabagiu and Miller and Moldovan 1999) in logic forms to QA systems. Consider again the simple scenario mentioned above. Even if we use resources such as WordNet (Fellbaum 1998) to find out that "to fly" means a person traveling in an aircraft, we still need human knowledge to figure out the effect of flying causes a person to be in a certain destination. The main difference of our proposed approach from other approaches is that we incorporate domain knowledge so that deep reasoning can be done in order to answer sophisticated questions.

It is evident that a proper representation and reasoning of the story is crucial to achieve accurate answers. AnsProlog, a logic programming language that is based on the answer set semantics (Gelfond and Lifschitz 1988, Gelfond and Lifschitz 1991), is a popular non-monotonic language that has gained wide acceptance due to its simplicity and expressiveness (Baral 2003). We propose the use of AnsProlog for representation and reasoning, together with Link Grammar (Sleator and Temperley 1993) to extract facts from the natural language text and WordNet (Fellbaum 1998) to disambiguate the meanings of the extracted verbs and nouns. We implemented a prototype that is based on a simple travel domain.

The outline of the paper is as follows: we first briefly introduce the basics of AnsProlog, Link Grammar and WordNet in the "Preliminaries" section. The system architecture of our question and answering system is then described in the "System Architecture" section, followed by a detailed description of the various components involved in the system. We then provide a simple scenario for queries in the "Queries" section to demonstrate our system. We then conclude in the "Discussion" section.

Preliminaries

In this section, we give a brief introduction to the syntax of AnsProlog (Baral 2003) and describe two important publicly available resources that are used in our question and answering system - Link Grammar (Sleator and Temperley 1993) and WordNet (Fellbaum 1998).

AnsProlog

An AnsProlog rule is of the form:

$$a \leftarrow a_0, \ldots, a_m, \textbf{not } a_{m+1}, \ldots, \textbf{not } a_n.$$

where a_i's are literals and **not** represents negation as failures. The intuitive meaning of the above rule is that if it is known that literals $a_0, \ldots, a_m$, are to be true and if $a_{m+1}, \ldots, a_n$ can be safely assumed to be false, then a must be

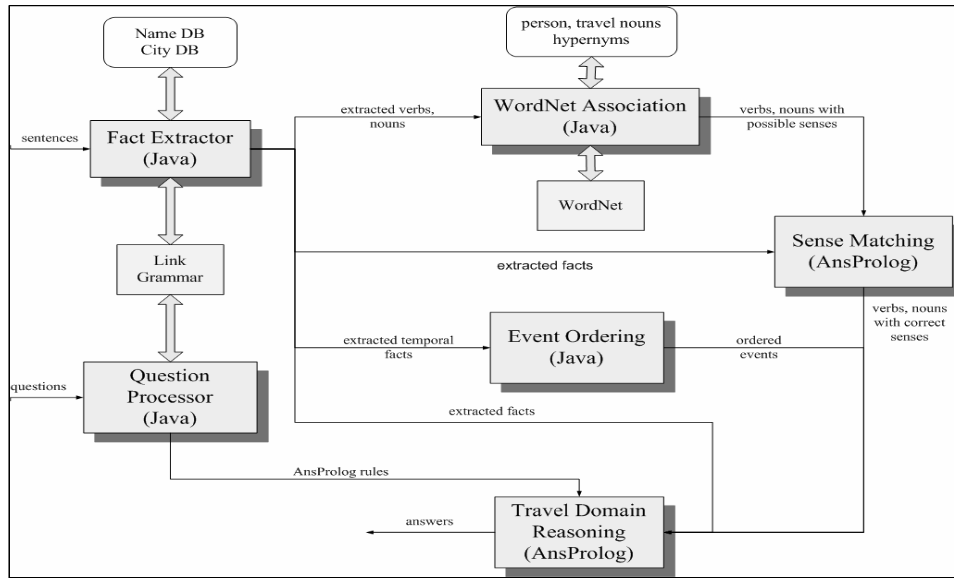

Figure 1 - System Architecture of our QA System with Deep Reasoning

true. A literal is defined as either an atom or an atom preceded by the symbol ¬.

Smodels (Niemelä and Simons 1997) is one of the popular answer set solvers for AnsProlog. In our QA system, Smodels is used to compute answer sets. The Smodels syntax of the above AnsProlog rule is as follows:

a `:-` a_0, …, a_m, **not** a_{m+1}, …, **not** a_n.

The '←' symbol is replaced by ":-", while the classical negation symbol ¬ is replaced by the symbol '-'.

Link Grammar

Typical natural language parsers, such as part of speech taggers, identify the part of speech of words for a sentence. The Link Grammar parser is a syntactic parser that parses English text based on the Link Grammar theory. Unlike a part of speech tagger, the parser outputs labeled links between pairs of words for a given input sentence. For instance, if word *a* is associated with word *b* by the link "S", *a* is identified as the subject of the sentence while *b* is the finite verb related to the subject *a*.

WordNet

WordNet is a lexical dictionary intended for the use of computers. The WordNet lexical database provides all possible senses for nouns, verbs, adjectives and adverbs organized in terms of word meanings rather than word forms (Miller et al 1993).

In our QA system, we utilize the hypernyms of nouns and verbs to disambiguate the various senses. Word *a* is a hypernym of word *b* if *a* has a "is_a" relation with *b*. WordNet 2.1 is used for our QA system.

System Architecture

In this section, we provide an overview of the architecture of our QA system and how each of the components interoperates. The input to the system is a list of sentences in natural language and the output is the answers for the questions asked with respect to the story.

The system is composed of several components: Fact Exactor, WordNet Association, Sense Matching, Event Ordering, Travel Domain Reasoning and Question Processor, as shown in figure 1. Given the input sentences, the component Fact Extractor utilizes Link Grammar to extract the necessary facts based on the links associated with the words in the sentences. The facts extracted by the Fact Extractor component are then passed along to the other components of the system.

In natural languages, it is usually the case that verbs and nouns have different meanings in different context. Therefore it is important to disambiguate the meanings of verbs and nouns in order to perform correct reasoning. To achieve this goal, the WordNet Association component assigns possible senses for the verbs and nouns extracted by the Fact Extractor component based on hypernyms provided by WordNet. Among the possible senses of verbs generated by the WordNet Association component, verbs are then further processed by the Sense Matching component by using the extracted facts to find the correct senses.

Ordering of events is important so that the story can be correctly represented. However, the actual time for the occurrence of events may not be mentioned in the story. The Event Ordering component orders and assigns events to various time points. With the extracted facts, verbs and

nouns with correct senses and ordered events, reasoning can then be done through the Travel Domain Reasoning component. The component consists of AnsProlog rules that describe inertia of fluents so as the effects and executability of actions. Smodels (Niemelä and Simons 1997) is used to as the answer set solvers.
The last component of our system is the Question Processor, which translates questions in natural language into AnsProlog rules automatically. The component is still in its infancy stage, so the majority of the queries are written manually in the form of AnsProlog rules.

Fact Extractor

The first step of our QA system is to extract facts in the form of AnsProlog from the sentences. This is done by parsing each sentence using the Link Grammar parser, so that an output showing links between pairs of words is produced. A simple algorithm is then used to generate AnsProlog facts based on the links. The algorithm works as follows:
Input: pairs of words associated with links produced by Link Grammar
Output: Facts in the form of AnsProlog

1. Suppose *ei* is the current event number and the event is in the *j*-th sentence. Form the facts *in_sentence(ei, j)* and *event_num(ei)*.
2. If word *a* is associated with word *b* through the link "S", then form the facts *event_actor(ei, a)* and *event_nosense(ei, b)*. If *a* appears in the name database, then form the fact *person(a)*.
3. If word *a* is associated with word *b* through the link "MV" and *b* is also associated with word *c* through the link "J", then form the fact *parameter(ei, b, c)*. If *c* appears in the city database, then form the fact *city(c)*.
4. If word *a* is associated with word *b* through the link "O", then form the facts *noun(b)* and *object(ei, b)*.
5. If word *a* is associated with word *b* through the link "ON" and *b* is also associated with word *c* through the link "TM", then form the fact *occurs(ei, b, c)* and *time(c)*.
6. If word *a* is associated with word *b* through the link "TY", then form the fact *occurs_year(ei, b)*.
7. If word *a* is associated with word *b* through the link "D", then form the fact *noun(b)*.

Example 1 Given the sentence "The train stood at the Amtrak station in Washington DC at 10:00 AM on March 15, 2005.", Figure 2 shows the corresponding output from Link Grammar.
The following facts are extracted based on the Link Grammar output:

```
event_num(e1).
in_sentence(e1,1).
event_actor(e1,train).
event_nosense(e1,stood).
parameter(e1,at,amtrak_station).
parameter(e1,in,washington_dc).
parameter(e1,at,t10_00am).
occurs(e1,march,15).
occurs_year(e1,2005).
person(john).
city(washington_dc).
city(new_york_city).
verb(stood).
noun(train).
noun(amtrak_station).
time(t10_00am).
```

□

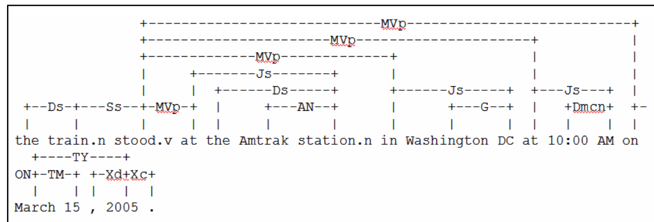

Figure 2 – Output of the Link Grammar Parser corresponding to the sentence "The train stood at the Amtrak station in Washington DC at 10:00 AM on March 15, 2005."

WordNet Association

The role of the WordNet Association component is to disambiguate the meanings of nouns and verbs. In the travel domain, it is essential to identify nouns that are in fact transportation or persons. Such identification is done through predefined lists of hypernyms for both transportation and persons. We called the predefined lists as root hypernyms. The root hypernyms for transportation consists of the words "travel", "public transport" and "conveyance". The word "person" is the only root hypernym for persons.
Together with the root hypernyms, each noun is first queried through WordNet to find its hypernyms. If one of the hypernyms matches the root hypernyms of transportation, then the noun is regarded as transportation. Similarly, if the hypernyms of a noun matches the root hypernyms of persons, then the nouns is regarded as a person. The following examples illustrate the idea:

Example 2 Given the noun "train", Figure 3 shows a partial list of hypernyms produced by WordNet. In sense 1, public transport is a hypernym of train, so the WordNet Association component outputs the fact `transportation(train)`.

```
Sense 1
train, railroad train
       => public transport
           => conveyance, transport
               => instrumentality, instrumentation
                   => artifact, artefact
                       => whole, unit
                           => object, physical object
                               => physical entity
                                   => entity
Sense 2
string, train
       => series
           => ordering, order, ordination
               => arrangement
                   => group, grouping
                       => abstraction
                           => abstract entity
                               => entity
```

Figure 3 – Hypernyms of the word "train" produced by WordNet

□

Example 3 Given the noun "conductor", Figure 4 shows a partial list of hypernyms produced by WordNet.
In sense 4, person is a hypernym of conductor, so the WordNet Association component outputs the fact `person(conductor)`.

□

The WordNet Association component performs a similar process on each verb extracted from a sentence by using the hypernyms of WordNet. Rather than relying on a list of predefined hypernyms as in the case of nouns, the component returns all possible senses of a given verb. Given the verb *v* and *v* has the hypernym *v'*, then the component returns the fact *is_a(v, v')*. This process is illustrated in example 4.

```
Sense 3
conductor
       => material, stuff
           => substance, matter
               => physical entity
                   => entity
Sense 4
conductor
       => collector, gatherer, accumulator
           => holder, bearer
               => capitalist
                   => person, individual, someone, somebody, mortal, soul
                       => organism, being
                           => living thing, animate thing
                               => object, physical object
                                   => physical entity
                                       => entity
                       => causal agent, cause, causal agency
                           => physical entity
                               => entity
```

Figure 4 – Hypernyms of the word "conductor" produced by WordNet

Example 4 Given the verb "stood", Figure 5 shows a partial list of hypernyms returned from WordNet.
In figure 4, the words "rest", "be", "resist", "fight", "contend" are the hypernyms of "stood". Therefore, the facts below are returned by the WordNet Association component:

```
is_a(stood,rest).
is_a(stood,be).
is_a(stood,resist).
is_a(stood,fight).
is_a(stood,contend).
```

□

```
12 senses of stand
Sense 1
stand, stand up
       => rest
          => be
       Also See-> stand up#1
Sense 2
stand
       => be
       Also See-> stand for#3; stand out#1; stand out#2; stand by#1; stand by
#3; stand firm#1
Sense 3
stand
       => be
Sense 4
stand, remain firm
       => resist, hold out, withstand, stand firm
           => fight, oppose, fight back, fight down, defend
               => contend, fight, struggle
```

Figure 5 – Hypernyms of the word "stood" produced by WordNet

Sense Matching

With the possible senses of verbs generated by the WordNet Association component, the correct senses are matched using the extracted facts related to the same event. AnsProlog rules are written to match the correct senses of verbs. The following rules are used for matching the correct senses of the verb "stood":

```
%% verb V means occupying a place or
%% location
event(E,be) :- event_actor(E,TR),
  is_a(V,be), parameter(E,at,C),
  event_nosense(E,V), parameter(E,at,T).

%% verb V means occupying a place or
%% location
event(E,be) :-  event_actor(E,TR),
  is_a(V,be), parameter(E,in,C),
  event_nosense(E,V), parameter(E,at,T).
```

The first rule says that if an event *E* involves an actor *TR* which is a kind of transportation, a city *C*, time *T* and the verb *V* has a hypernym as "be", then the action involved in *E* has the same meaning as the verb "be". In other words, the verb *V* has the meaning of "be" in the context of the event *E*. The second rule works similarly, other than city *C* has to be associated with the preposition "in" rather than "at" as in the first rule.
As another example of how the Sense Matching component works, the following rule is used to match the correct senses of a verb that has the meaning of "enter":

```
%% verb V means to get on board of
%% some kind of transportation
event(E,enter) :- event_actor(E,P),
    is_a(V,enter), event_nosense(E,V),
    object(E,TR),parameter(E,at,T).
```

The intuitive meaning of the above AnsProlog rule says that verb *V* has the meaning of "enter" if event *E* has person *P* as the actor and a transportation *TR* and time *T* are involved in event *E*.

Event Ordering

The events extracted are needed to be ordered based on the time specified in each of the events. However, in the description of a story, it is rarely the case that all events mentioned have explicit mention of time. To assign events with unknown time, we use ordering of sentences as an assumption. Suppose event *e1* is known to happen at time *t1* and the story did not mention when event *e2* occurs. Among the sentences, *e1* is mentioned before *e2*. According to the ordering of the sentences, we can assume that *e2* happens after *t1*. The following is a simple algorithm used to order and assign events to timepoints:

Let $E = \{e_1,\ldots,e_n\}$ be a set of events and each event e_k is associated with an actual time t_k, where t_k is in an ordered list of time T. Suppose the actual time of event e_j is not known, then t_j = "unknown" and t_j is placed at the end of the list T.

1. For each event $e_k \in E$ associated with actual time t_k, create timepoints tp_{2k} and tp_{2k+1} and map e_k to timepoint tp_{2k}.
2. Create an extra timepoint tp_1.
3. Create a fact *timepoint*(tp_1, *before*, t_1) to indicate that timepoint tp_1 refers to before time t_1.
4. Create facts *timepoint*(tp_{2k}, *at*, t_k) and *timepoint*(tp_{2k+1}, *after*, t_k).
5. Iterate through all sentences in the order of the given sentences. If event e_k appears in the *i*-th sentence and e_k is associated with time t_k = "unknown", then
    a. Find an event e_j such that e_j is associated with $t_j \neq$ "unknown" and e_j appears before the *i*-th sentence.
    b. Maps e_k to timepoint tp_{2j+1}.

The following example is used to illustrate the idea of event ordering and assignment:

Example 5 Let the predicate *in_sentence*(e_k, *i*) imply that event e_k appears in the *i*-th sentence and let the predicate *parameter*(e_k, *at*, t_k) represent event e_k occurs at actual time t_k. Suppose we have the following facts extracted by the Fact Exactor component:

parameter(e1, at, t10_00am).
in_sentence(e1, 1).
in_sentence(e2, 2).
parameter(e3, at, t10_30am).
in_sentence(e3, 3).

With the above facts, $E = \{e_1, e_2, e_3\}$ and T = <*t10_00am, t10_30am, unknown*>. The following set of timepoints is created using steps 1 and 2:

timepoints = $\{tp_1, tp_2, tp_3, tp_4, tp_5\}$, and e_1 is mapped to tp_2 and e_3 is mapped to tp_4.

In steps 3 and 4, the following facts are created:

timepoint(tp1, before, t10_00am).
timepoint(tp2, at, t10_00am).
timepoint(tp3, after, t10_00am).
timepoint(tp4, at, t10_30am).
timepoint(tp5, after, t10_30am).

In step 5, since e_2 is associated with unknown time and e_1 appears ahead of e_2 among the sentences, e_2 is mapped to tp_3.

□

Travel Domain Reasoning

The Travel Domain Reasoning component utilizes the output of other components described earlier for reasoning. The component is written in AnsProlog rules and the rules can be grouped into three categories: action rules, fluent-action rules and general rules.

Action Rules

Action rules are rules that describe the occurrences of actions based on the extracted facts and the executability of actions. Some of the rules in this category are shown below:

```
%% transportation TR is at location
%% number LN at timepoint TP
o(be_at(TR,LN),TP) :-
   event_actor(E,TR), event(E,be),
   parameter(E,at,C), maps_to(LN,C),
   is_associated(E,TP).
```

The above rule describes when the action for transportation *TR* to be at location *LN* takes place. Such an action can occur at timepoint *TP* only if event *E* has a transportation *TR* as the actor, the action is about being in a city *C* and *E* is associated to timepoint *TP*.

Similarly, the rules below describe the occurrence for person *P* to enter transportation *TR* at timepoint *TP*. The last part of the first rule describing the action "enter" says that the action "enter" can take place at timepoint *TP*, unless it is known that the action cannot be taken place at timepoint *TP*. This happens when person *P* and transportation *TR* are in different locations, which is expressed in the second rule. Such kind of rule is called the executability of actions.

```
%% person P enters transportation TR at
%% timepoint TP
o(enter(P,TR),TP) :-
  event_actor(E,P), object(E,TR),
  event(E,enter), is_associated(E,TP),
  not -o(enter(P,TR),TP).

%% Person P cannot enter train TR at
%% timepoint TP if P and TR are in
%% different locations at timepoint TP
-o(enter(P,TR),TP) :- h(p_at(P,LN),TP),
  h(t_at(TR,LN1),TP),LN != LN1.
```

Fluent-Action Rules

Fluent-Action rules describe the direct and indirect effects of actions on the fluents or properties of the world. In other words, the rules describe what will happen when an action is taken place. Below are some of the fluent-action rules:

```
%% person P is at location number LN at
%% timepoint TP
h(p_at(P,LN),TP) :- o(be_at(P,LN),TP).
%% transportation TR is at location
%% number LN at timepoint TP
h(t_at(TR,LN),TP) :-
     o(be_at(TR,LN),TP).
```

The above rules describe the state of the world when the action "be_at" occurs, the fluent person *P* or transportation *TR* is at location number *LN* is true at timepoint *TP*. The rule below describes the effect of the occurrence of action "enter", and states that person *P* enters transportation *TR* at timepoint *TP* implies that *P* is in *TR*.

```
%% person P is in transportation TR at
%% timepoint TP
h(in(P,TR),TP) :- o(enter(P,TR),TP).
```

For a person to be in a transportation, we need to capture the intuition that the person should be in wherever location the transportation currently is at. Likewise, if a transportation in a location, we can safely say that the person on the transportation is also not in that particular location.

```
%% person P is at the same location as
%% transportation TR if P is in TR
h(p_at(P,LN),TP) :-
  h(t_at(TR,LN),TP), h(in(P,TR),TP).
-h(p_at(P,LN),TP) :-
  -h(t_at(TR,LN),TP), h(in(P,TR),TP).
```

General Rules

An important property about fluents in the action theory is to capture a fluent *f* at timepoint *TP* remains true at timepoint *TP*+1 unless we know that it cannot be true at timepoint *TP*+1. Likewise, fluent *f* at timepoint remains false at timepoint *TP*+1 unless something causes *f* to be true at *TP*+1. This property is known as inertia of fluents. The rules below capture this important property:

```
%% Inertia of fluents FL at timepoints
%% TP and TP+1
h(FL,TP+1) :-
  h(FL,TP), not -h(FL,TP+1).
-h(FL,TP+1) :-
 -h(FL,TP), not h(FL,TP+1).
```

Question Processor

The last component of our question and answering system is the Question Processor. The goal of the Question Processor component is to translate questions in natural language into AnsProlog queries (or rules) so that we can get the answers using the extracted facts of the stories and the reasoning component.

The first step of the component is to parse a given question using the Link Grammar parser. The links determine what type of question is being asked and what sort of AnsProlog "templates" should be used to form a query. We illustrate this step with the example below.

Example 6 Given the input question "Where was the train on March 15?", Figure 6 shows the corresponding output from the Link Grammar Parser.

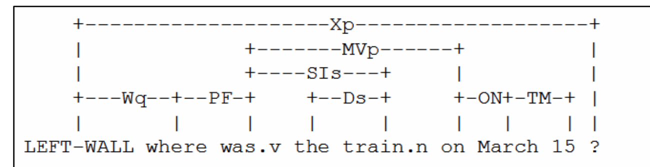

Figure 6 – Output of the Link Grammar Output corresponding to the question "Where was the train on March 15?"

The first step is to recognize what type of question is being asked. The word "where" indicates that the question is about asking for a city or a place. The first part of the translation is as below.

```
query0(C) :-
 h(t_at(ACTOR,LN),TP), maps_to(LN,C),
 event_actor(E,ACTOR),
```

The second step is to identify what "ACTOR" is referred to in this question. The link "SI" indicates that "train" is the actor of the question.

```
ACTOR = train,
```

The third step is to figure out the conditions of the question. In this case, the links "ON" and "TM" indicate the condition is about a particular date. So the following is added to the translation:

```
occurs(E,MO,DAY), is_associated(E,TP),
```

To recognize what "MO" and "DAY" are, the link "TM" indicates that

```
MO=3, DAY=15.
```

Therefore the question is translated into the AnsProlog rule

```
query0(C) :-
  h(t_at(train,LN),TP), maps_to(LN,C),
  event_actor(E,train), occurs(E,3,15),
  is_associated(E,TP).
```

□

Currently, WordNet is not utilized to identify if the actor of the question is a person or a transportation. In addition, the question "where" can have the answer of a city or a particular place. Therefore, the following AnsProlog rules are generated for the question "Where was the train on March 15?":

```
%%% Where was the train on March 15?
query0(C) :- h(t_at(train,LN),TP),
  maps_to(LN,C), event_actor(E,train),
  occurs(E,3,15), is_associated(E,TP).
query0(C) :- h(p_at(train,LN),TP),
  maps_to(LN,C), event_actor(E,train),
  occurs(E,3,15), is_associated(E,TP).
query0(W) :- in(W,C),
  h(t_at(train,LN),TP), maps_to(LN,C),
  event_actor(E,train), occurs(E,3,15),
  is_associated(E,TP).
query0(W) :- in(W,C),
  h(p_at(train,LN),TP), maps_to(LN,C),
  event_actor(E,train), occurs(E,3,15),
  is_associated(E,TP).
```

To show that the process of translation questions into AnsProlog rules can be generalized to other questions about "where", example 7 is used to illustrate the idea.

Example 7 Given the input question "Where was the train on March 15?", Figure 7 shows the corresponding output from the Link Grammar Parser.

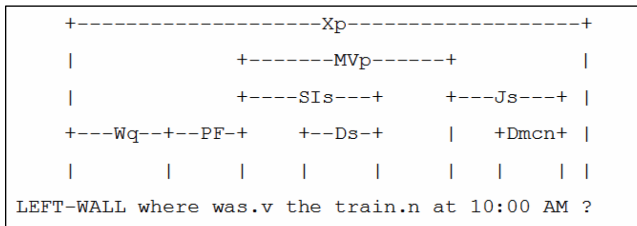

Figure 7 – Output of the Link Grammar Output corresponding to the question "Where was the train at 10:00 AM?"

Similar to example 6, we have the following translation that utilizes "where" and "train" being an actor:

```
query1(C) :-
   h(t_at(train,LN),TP), maps_to(LN,C),
   event_actor(E,train),
```

The link "Dmcn" indicates that the question has the condition about time. So the rest of the query is `timepoint(TP,at,T), T=t10_00am.`

Therefore the question is translated into the AnsProlog rules:

```
%%% Where was the train at 10:00 AM?
query1(C) :- h(t_at(train,LN),TP),
  maps_to(LN,C), event_actor(E,train),
  timepoint(TP,at,T), T=t10_00am.
query1(C) :- h(p_at(train,LN),TP),
  maps_to(LN,C), event_actor(E,train),
  timepoint(TP,at,T), T=t10_00am.
query1(W) :- in(W,C),
  h(t_at(train,LN),TP), maps_to(LN,C),
  event_actor(E,train),
  timepoint(TP,at,T), T=t10_00am.
query1(W) :- in(W,C),
  h(p_at(train,LN),TP), maps_to(LN,C),
  event_actor(E,train),
  timepoint(TP,at,T), T=t10_00am.
```

□

Queries

To evaluate our question and answering system, we use the following simple story about John traveling on a train:

> The train stood at the Amtrak station in Washington DC at 10:00 AM on March 15, 2005.
>
> The train was scheduled to depart for New York City at 10:30 AM and arrive at 1:30 PM on March 15.
>
> John arrived at the Amtrak station at 10:15 AM.
>
> John boarded the train at 10:20 AM and handed the ticket to the conductor.
>
> The conductor punched the ticket.
>
> John sat by a window.
>
> The train left the Amtrak station on time.

Using the questions shown in examples 6 and 7, the answers are as follows:

```
%% Q0: Where was the train on March 15?
query0(new_york_city).
query0(washington_dc).
query0(amtrak_station).

%% Q1: Where was the train at 10:00 AM?
query1(washington_dc).
query1(amtrak_station).

%% Q2: Where was the train at 10:15 AM?
query2(washington_dc).
query2(amtrak_station).
```

It is interesting to notice that the story does not explicitly mention where the train is at 10:15 AM. Since we know that the train is at the Amtrak station in Washington DC at 10 AM, using inertia of fluents, we can conclude that the train is still at the Amtrak station in Washington DC. This is indicated by the answers for question Q2.
The following questions, which can be answered without domain knowledge, are encoded manually as AnsProlog rules:

```
%% Q3: What was the train's
```

```
%% destination?
dest(train,C) :- event_actor(E,train),
  event(E,leave), parameter(E,for,C).

%% Q4: What time did the train depart?
depart_at(train,T) :-
  h(t_at(train,LN),TP), dest(train,C),
  maps_to(LN,C),
  -h(t_at(train,LN),TP+1),
  timepoint(TP,at,T).

%% Q5: What date did the train leave?
depart_on(train,M,D) :-
  h(t_at(train,LN),TP), dest(train,C),
  maps_to(LN,C),
  -h(t_at(train,LN),TP+1),
  timepoint(TP,at,T),
  is_associated(E,TP), occurs(E,M,D).

%% Q6: What date did John arrive in
%% New York City?
get_to_NYC(john,M,D) :-
  not h(p_at(john,LN),TP),
  maps_to(LN,new_york_city),
  h(p_at(john,LN),TP+1),
  is_associated(E,TP+1), occurs(E,M,D).

%% Q7: If John arrived in New York City
%% as scheduled, what time did he
%% arrive? What date?
at_NYC(john,T) :-
  maps_to(LN,new_york_city),
  h(in(john,TR),TP+1),
  not h(t_at(TR,LN),TP),
  h(t_at(TR,LN),TP+1),
  timepoint(TP+1,at,T).

on_NYC(john,M,D) :-
  maps_to(LN,new_york_city),
  h(in(john,TR),TP+1),
  not h(t_at(TR,LN),TP),
  h(t_at(TR,LN),TP+1),
  is_associated(E,TP+1), occurs(E,M,D).

%% Q8: Who punched John's ticket?
who_punched(P) :-
  o(punch(P,ticket),TP),
  -h(own(john,ticket),TP+1).
```

The corresponding answers to the above queries are:

```
%% Answer to Q3
dest(train,new_york_city).

%% Answer to Q4
depart_at(train,t10_30am).

%% Answer to Q5
depart_on(train,3,15).

%% Answer to Q6
get_to_NYC(john,3,15).

%% Answers to Q7
on_NYC(john,3,15).
at_NYC(john,t1_30pm).

%% Answer to Q8
who_punched(conductor).
```

Consider the following queries that cannot be answered directly from the story:

```
%% Q9: If John did not arrive at the
%% Amtrak station after 10:30 AM,
%% would he have boarded the train?
make_it(yes) :- john_at(W,TP1),
  in(W,C), h(t_at(train,LN),TP1),
  maps_to(LN,C).
make_it(no) :- not make_it(yes).
john_at(amtrak_station,TP+1) :-
  timepoint(TP,after,t10_30am).
```

The answer to Q9 is *make_it(no)*, due to the fact that the train leaves at 10:30. So John would not able to board the train if he arrived after 10:30 AM.

```
%% Q10: When did the conductor punch
%% the ticket?
when_punch(EP,T) :-
  event_actor(E,conductor),
  o(punch(conductor,ticket),TP),
  is_associated(E,TP),
  timepoint(TP,EP,T).
```

Notice that in the story, it is not mentioned that when the conductor punched the ticket. However, based on sentence ordering, we can assume that it happened after John boarded the train at 10:20 AM. So the answer is *when_punch(after, t10_20am)*.

Consider the following hypothetical queries that require domain knowledge about travel:

```
%% Q11: If John did not have a ticket,
%% can he board the train?
-h(own(john,ticket),0).
board_train(john,yes) :-
   h(in(john,train),TP).
board_train(john,no) :-
   not board_train(john,yes).
```

In order to answer Q11, the system needs extra knowledge about the fact that it is required to have a ticket to board a train. Such knowledge is not mentioned in the story and we incorporate a rule describing the knowledge in AnsProlog as part of the general travel domain knowledge. Our QA system returns *board_train(john, no)*, indicating that john cannot board the train.

As another example of using domain knowledge to answer questions, consider the following question:

```
%% Q12: If the train was an Amtrak
%% train, when would he arrive?
amtrak(train).
when_arrive(TR,EP,T) :-
  h(t_at(TR,1),TP), timepoint(TP,EP,T).
```

A train that leaves on time normally arrives on time. For Amtrak trains, they usually arrive late even when they leave on time. Here we need a default rule that describes trains normally arrive on time and treat Amtrak trains as abnormal trains. With this extra knowledge, our system returns the answer *when_arrive(train, after, t1_30pm)*, indicating that the train arrives late rather than at 1:30 PM, once we know that the train is an Amtrak train.

Discussion

We described a simple approach to question and answering with deep reasoning that utilizes AnsProlog for representation and reasoning, together with Link Grammar for fact extraction and WordNet for disambiguating verbs and nouns. This approach is different from previous approaches in the sense that we use AnsProlog to express certain domain knowledge that is required for deep reasoning. Using the simple traveling scenario and queries described in the paper, we showed that such sophisticated questions cannot be answered without the use of domain knowledge about travel and deep reasoning.

As future work, we need to expand the AnsProlog rules for the travel domain and be able to translate different types of questions into AnsProlog rules for our Question Processor.

References

Baral, C. 2003. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press.

Fellbaum, C. eds. 1998. *WordNet: An Electronic Lexical Database*. MIT Press.

Gelfond, M. and Lifschitz, V. 1988. *The Stable Model Semantics for Logic Programs*. International Symposium on Logic Programming, pp. 1070 - 1080, MIT Press.

Gelfond, M. and Lifschitz, V. 1991. *Classical Negation in logic programs and disjunctive databases*. New Generation Computing, pp. 365 - 387.

Harabagiu, S., Miller, A., Moldovan, D. 1999. *Wordnet 2 – a morphologically and semantically enhanced resource*. In Proceedings of SIGLEX-99, pp. 1-8.

Harabagiu, S., Pasca, M. and Maiorano, S. 2000. *Experiments with Open-Domain Textual Question Answering*, in Proc. of COLING-2000, August 2000, Saarbruken Germany, pp. 292-298.

Harabagiu, S. 2001. *Just-In-Time Question Answering*. Invited talk in Proc. of the Sixth Natural Language Processing Pacific Rim Symposium (NLPRS-2001), Tokyo, Japan, pp. 27-34.

Miller, G., Beckwith, R., Fellbaum, C., Gross, D., Miller, K. 1993. *Introduction to Wordnet: An Online Lexical Database*, http://www.cogsci.princeton.edu/~wn/papers.shtml.

Niemelä, I. and Simons, P. 1997. *Smodels - An Implementation of the Stable Model and Well-Founded Semantics for Normal LP*. Logic Programming and Non-monotonic Reasoning, pp. 421 – 430, Springer Verlag.

Pasca, M. 2000. *Open-domain factual answer extraction*. Technical Report, Southern Methodist University.

Pasca, M. and Harabagiu, S. 2001. *The Informative Role of WordNet in Open-Domain Question Answering*, in Proceedings of the NAACL 2001 Workshop on WordNet and Other Lexical Resources: Applications, Extensions and Customizations, Carnegie Mellon University, Pittsburgh PA, pp. 138-143.

Rus, V. 2002. *Logic forms for Wordnet Glosses*. PhD thesis, Southern Methodist University.

Sleator, D. and Temperley, D. 1993. *Parsing English with a Link Grammar*. Third International Workshop on Parsing Technologies.

Vicedo, J. L. 2000. *A semantic approach to question answering systems*. In Proceedings of Text Retrieval Conference (TREC-9), NIST, pp. 440 – 445.