An Answer Set Programming based Approach to Representing and Querying Textual Knowledge

Dhruva Pendharkar and **Gopal Gupta**
The University of Texas at Dallas
Richardson, TX 75080, USA
dhruva.pendharkar@gmail.com, gupta@utdallas.edu

Abstract

An approach based on answer set programming (ASP) is proposed in this paper for representing knowledge generated from natural language text. Knowledge in the text is modeled using a Neo Davidsonian-like formalism, represented as an answer set program. Relevant common sense knowledge is additionally imported from resources such as WordNet and represented in ASP. The resulting knowledge-base can then be used to perform reasoning with the help of an ASP system. This approach can facilitate many natural language tasks such as automated question answering, text summarization, and automated question generation. ASP-based representation of techniques such as default reasoning, hierarchical knowledge organization, preferences over defaults, etc., are used to model common-sense reasoning methods required to accomplish these tasks. In this paper we describe the CASPR system that we have developed to automate the task of answering natural language questions given English text. CASPR can be regarded as a system that answers questions by "understanding" the text and has been tested on the SQuAD data set, with promising results.

Introduction

The goal of artificial intelligence is to build systems that can exhibit human-like intelligent behavior. Decision making and the ability to reason are important attributes of intelligent behavior. Hence, machines possessing artificial intelligence must be capable of performing automated reasoning as well as responding to changing environment (for example, changing knowledge). To exhibit such an intelligent behavior, a machine needs to understand its environment as well be able to interact with it to achieve certain goals. Classical logic based approaches have traditionally been used to build automated reasoning systems but have not lead to systems that can be called truly intelligent. Humans, arguably, do not use classical logic in their day to day reasoning tasks. They considerably simplify their burden of reasoning by using techniques such as defaults, exceptions, and preference patterns. Also, humans use non-monotonic reasoning and can deal with incomplete information (Gelfond and Kahl 2014; Jonson-Laird 2009). All these features need to be built into an intelligent system, if we want to simulate human-like intelligence. It has been shown that common-sense reasoning can be realized via a combination of (stable model semantics-based) negation as failure and classical negation (Baral 2003) (Gelfond and Kahl 2014) in ASP. ASP is a well-developed paradigm and has been applied to solving problem in planning, constraint satisfaction and optimization. There are comprehensive, well known implementations of ASP such as CLASP (Gebser et al. 2010) and DLV (Alviano et al. 2011). Scalable implementations of ASP that support predicates (i.e., do not require grounding) and are query-driven, such as s(ASP), have also been developed (Marple, Salazar, and Gupta 2017; Arias et al. 2018). ASP is also well suited for representing knowledge and for modelling common-sense reasoning. Most of the knowledge resources available today are in the form of unstructured data, either in the form of written documents, or information present online. With the help of formalisms such as the event calculus (Kowalski and Sergot 1989), the situation calculus (Van Harmelen, Lifschitz, and Porter 2008), or neo-Davidsonian logic (Davidson 1984), ASP could prove to be a helpful paradigm to represent this textual knowledge and reason with it.

In this paper we propose a system called CASPR (Commons-sense ASP Reasoning) to automatically convert textual knowledge into ASP programs, and use it to answer questions coded as ASP queries. The problem of converting natural language text into ASP is challenging enough, however, even if we succeed in this translation task, the resulting knowledge is not enough to answer questions to the level that a human can. When humans read a passage, we automatically draw upon a large amount of common sense knowledge that we have acquired over the course of years in understanding the passage and in answering questions related to the passage. An automated QA system ought to do the same. CASPR, thus, resorts to resources such as WordNet (Miller 1995), that encapsulate some of the common-sense knowledge, to augment the knowledge derived from the text. CASPR also allows users to add common sense knowledge—coded in ASP—manually as well.

CASPR runs on the s(ASP) answer set programming system. The s(ASP) system (Marple, Salazar, and Gupta 2017) is a query-driven predicate ASP system that is sacalable, in that it can run answer set programs containing predicates

with arbitrary terms. The s(ASP) system is query-driven and does not require grounding, a crucial feature needed for building large-scale natural language-based KR applications using ASP. Since s(ASP) is query-driven, it can provide a justification for the query being processed, so it is possible to give the reasoning behind an answer to a question in CASPR.

Our research makes several contributions: (i) it shows that with the help of pioneering systems such as s(ASP), it is possible to build practical NLP applications that rely on "text understanding"; and, (ii) traditional problems of Natural Language Understanding such as word sense disambiguation can be solved quite elegantly with ASP.

System Architecture

CASPR is composed of two main components or sub systems viz. the Knowledge Generation System and the Query Generation System. Both these systems function independently of each other. The architecture, as illustrated in Figure 1, comprises of a common resource framework that is shared by both these systems.

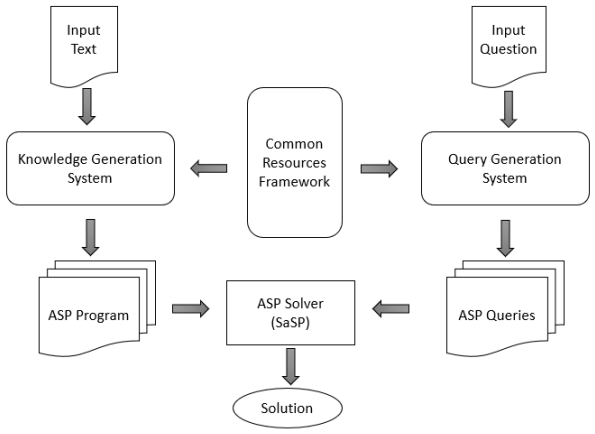

Figure 1: System Architecture

The Common Resource Framework consists of Natural Language Processing tools such as Stanford Core NLP Tools, WordNet API as well as modules for pre-processing incoming text. The Knowledge Generation System is mainly responsible for extracting knowledge from a natural language text. For extracting the knowledge from text, this component uses Stanford NLP tools like the POS Tagger (Toutanova and Manning 2000; Toutanova et al. 2003), Stanford Dependency Parser (Chen and Manning 2014), and the Stanford NER Tagger (Finkel, Grenager, and Manning 2005) to gain more information about the input text. Apart from these resources it also taps into the vast information that is provided by WordNet (Miller 1995) and extracts information from it, converting it into an answer set program. WordNet provides significant amount of common sense knowledge about nouns. For verbs, at present, there are very few digital resources available that are similar to WordNet. Common sense knowledge for verbs, thus, has to be modelled manually as an answer set program, and added to the common resource framework.

The Query Generation System automatically translates the question to an ASP query that can be executed against the ASP-coded knowledge-base generated from the textual passage augmented with common sense knowledge. Execution is performed using the s(ASP) system. The solution found represents an accurate answer to the question. The query obtained from the natural language question is a conjunction of multiple sub-goals. If the query fails, then some of the sub-goals are systematically removed and the query re-executed to find less accurate answers. These less accurate answers are reported too, along with the level of accuracy (likely, possible, guess).

Knowledge Representation

The Stanford Dependency Parser is used to parse the pre-processed text and a semantic graph generated using the Stanford Typed Dependencies representation (De Marneffe and Manning 2008; De Marneffe et al. 2014).

Example 1. "NASA carried out the Apollo program."
Following is the Stanford Dependency (SD) representation:

nsubj(carried-2,NASA-1), root(ROOT-0,carried-2), compound:prt(carried-2,out-3), det(program-6,the-4), compound(program-6,Apollo-5),dobj(carried-2,program-6)

These dependencies map straightforwardly onto a directed graph representation in which words in the sentence are nodes in the graph and grammatical relations are edge labels. In English, most event mentions correspond to verbs and most verbs are triggers to events. Although this is true in most cases there are other word groups that can trigger events as well. The different verbs in the sentence thus, define various events that take place in the sentence and how these events are connected to each other. Let us consider a more complicated example as follows.

Example 2. "Miitomo, which Nintendo introduced globally in 2016, features the company's, Mii, avatar-system and lets the users communicate by exchanging personal information such as favorite movies."

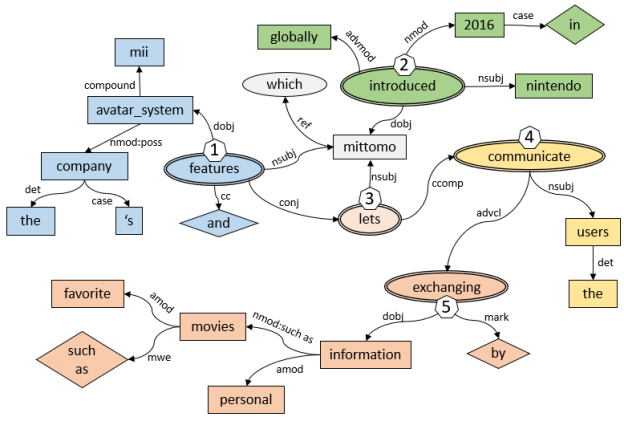

Figure 2: Semantic graph representing event regions

Figure 2 shows how various words in the example passage are connected to each other in the sentence. The main verbs of the sentence are "*feature*" and "*let*" connected by a coordinating conjunction. The verbs in the Figure 2 represent the head of events in the sentence and are marked using event IDs. The various color regions denote the rough boundaries of these event regions. The semantic graph along with the event regions are used to create facts and rules.

Predicate Generation

Knowledge is represented using a collection of pre-defined predicates, following the neo-Davidsonian approach (Davidson 1984). Some of the predicates capture specialized concepts such as *abbreviation*, *start_time* etc., whereas others are more generic like *mod*, *event* and so on. The generic predicates that convey information are explicitly present in the text, whereas all others model implicit information. Note that it is important to keep this predicate representation as simple as possible, so that individual pieces of knowledge can be composed using common sense reasoning patterns. These reasoning patterns have to be kept very simple as well. Otherwise, we run the risk that we may have appropriate knowledge in our knowledge-base to answer the given question, but we fail in answering it because we are unable to compose that knowledge due to complexity of its representation. Following are some of the important predicates that have been used by CASPR:

Event Predicate: The event predicate defines an event that happens in the sentence. The verb marks the head of the event predicate. In general, an event consists of certain actors and participants taking part in the event. The *event* predicate consists of the various actors (doers) and participants involved in the event. The signature of the event predicate can be given as follows:

event (event_id, trigger_verb, actor, participant)

where the *event_id* is an integer that uniquely identifies that event in the paragraph. The *trigger_verb* denoted in the event predicate is the *lemma*, i.e., the stem word, of the actual word used in the sentence. The actors in the event predicate are the subjects found to the trigger_verb in the sentence. Subjects in the sentence can be found with the help of dependencies like *nsubj* and *nsubj:xsubj*. Just as actors can be obtained from the subject of the sentence, the participants can be determined from direct object dependency (*dobj*).

Example 3. "The American_Football_Conference's (AFC) champion team, Denver_Broncos, defeated the National_Football_Conference's (NFC) champion team, Carolina_Panthers, by 24_10 to earn AFC third Super_Bowl title"

event (1, defeat, denver_broncos, carolina_panthers)
event (2, earn, afc, title)

Here we can get richer information about the event by generating duplicate event predicates for each modifier for the actor as well as the participants involved in the event. To generate such event predicates, we use the *amod* or *nummod* dependencies for the actors and the participants and create compound atoms from the modifiers and their governors. Consider the following duplicate event predicate:

event (2, earn, afc, third_super_bowl_title)

Note that in absence of information, the default value for the actor and the participant field maybe null. A null value indicates that either the term is absent for the event or the system was not able to determine it.

Property Predicate: The property predicate elaborates on the properties of the modified noun or verb. The modifier in this case is generally a prepositional phrase in the sentence. A property predicate is coupled with an event and describes the modification only for that event. The signature for the property predicate is as follows:

_property(event_id,modified_entity,preposition,modifier)

The *event_id* of this predicate should belong to one of the events in the context. The *modified_entity* is the head of this predicate and can be a noun or a verb. The preposition is the dependent of the *case* relation with the modifier as the governor and the modifier can be found using the nominal modifier or the *nmod* relation from the dependency relations.

Example 4. "The game was played on February 7 2016, at Levis_Stadium, in the San_Francisco_Bay_Area, at Santa_Clara in California"

_property (2, play, on, 'february_7_2016')
_property (2, play, at, levis_stadium)
_property (2, play, in, san_francisco_bay_area)
_property (2, play, at, santa_clara)
_property (2, santa_clara, in, california)

From the above property predicates we can get more information about the *modified_words*, *play* and *santa_clara*.

Modifier Predicate: The modifier predicate is used to model the relationship between adjectives and the nouns they modify and between verbs and their modifying adverbs. It has the following signature:

_mod (modified_word, modifier_word)

The *modified_word* can be a noun or a verb. In case of the noun the *modifier_word* is an adjective given by the *amod* relation; if it is a verb then the modifier_word is an adverb given by the *advmod* relation.

Example 5. "The Amazon_rainforest, also known in English as Amazonia or the Amazon_Jungle, is a moist broadleafed forest that covers most of the Amazon_basin of South_America."

_mod (forest, broadleafed)
_mod (forest, moist)
_mod (know, also)

Possessive Predicate: The possessive predicate is used to model the genitive case in English. It is used to show possession or a possessive relation between two entities in the sentence. It has the following signature:

_possess(possessor, possessed)

Such a relation is generally present in nouns using the possessive case ('s). The possessor as well as the possessed are nouns in the sentence. The *_possess* relation can be extended towards appositional modifiers of the possessed.

Example 6. "The American_Football_Conference's (AFC) champion team, Denver_Broncos, defeated the National_Football_Conference's (NFC) champion team, Carolina_Panthers, by 24_10 to earn AFC third Super_Bowl title"

_possess (american_football_conference, team)

_possess (national_football_conference, team)
_possess (american_football_conference, denver_broncos)
_possess (national_football_conference, carolina_panthers)

From the above sentence we can generate the first two predicates, as they fall under the genitive case rule, using the *nmod:poss* dependency relation, but the next two facts need to be extended using the appositional modifier relation (*appos*).

Instance Predicate: The instance predicate models the concept of an instance. As an example, red is an instance of a color. We can model this fact as *color(red)* or *_is(red,color)*. The latter is preferred. The signature of the _is predicate can be given as follows

_is (instance, concept)

Here the instance as well as the concept are nouns having the relationship of instance_of. To model this relationship, we can use the *cop* dependency along with its related *nsubj*.

Example 7. "Nikola_Tesla was a serbian-american inventor, electrical engineer, mechanical engineer, physicist, and futurist"

_is (nikola_tesla, inventor).
_is (nikola_tesla, serbian_american_inventor).

In the above example we see that the verb (*is*) is associated with other concepts like engineer, physicist, and futurist using the conjunction. Thus, we can extend the definition of the instance predicate to also include these other facts:

_is(nikola_tesla,electrical_engineer),
_is(nikola_tesla, futurist),
....

Adding these facts makes the knowledge base richer which is now able to infer many other things about the passage. Another case, where we can generate the instance predicate is in cases where multi-word expressions like such as, or like are used to compare two concepts to be equivalent. Details are omitted due to lack of space.

Relation Predicate: The relation predicate is used to connect two concepts in events. This predicate is generated to model mainly two relations, dependent clauses and conjunctions, and has the signature:

_relation(independent_entity,dependent_clause_id,relation_type)

Dependent clauses can be of two types depending upon their governors i.e. adjective clauses or adverb clauses. Adjective clauses are dependent clauses that modify a noun, whereas adverb clauses modify a verb. The *independent_entity* defined in the signature can be either the noun that is modified or the event ID of the verb that is modified. The *dependent_clause_id* always refers to the ID of the verb that is the head of the dependent clause. The *relation_type* define the relation between the dependent and the independent clause. This system recognizes 3 types of relations viz. *_clause*, *_clcomplement* and *_conj*.

A relation with an adverb clause can be found out using the *advcl* dependency. Similarly, the adjective clause can be modeled using the *acl* dependency relation. One of the other relations is _clcomplement. This type of relation models both the clausal complement (*comp*) as well as the open clausal complement (*xcomp*). Such a relation is used to search for the subject of the dependent clause.

Example 8. "The American_Broadcasting_Company (ABC), stylized in the network's logo as ABC since 1957, is an American commercial broadcast television network"

_relation (american_broadcasting_company, 1, _clause)
event (1, stylize, null, null)

Example 9. "The ideal thermodynamic cycle used to analyze the process is called the Rankine_Cycle"

_relation (1, 2, _clcomplement)
event (1, use, null, null)
event (2, analyze, null, process)

Conjunctions are words that connect two or more clauses. The relation predicate is also used to model this relation between any two clauses. An example of such a predicate is given below.

Example 10. "Water heats and transforms into steam within a boiler operating at a high pressure"

_relation (2, 3, _conj)
event (2, heat, water, null)
event (3, transform, water, null)

Named Entity Predicate: CASPR uses the Named Entity Tagger (Finkel, Grenager, and Manning 2005) to get information about entities in the text. The Named Entity Tagger marks various classes like LOCATION, PERSON, ORGANIZATION, MONEY, PERCENT, TIME in the text. We make use of these tags to generate facts of the form *concept(instance)*. These facts together with the rules generated by the ontology help in reasoning about the text. Details are omitted due to lack of space, but can be found in (Pendharkar 2018).

Special Predicates: Special predicates have been used in the system, to model concepts that are patterns and are understood by humans implicitly. As grammatical relations in the sentence do not convey the meaning of these concepts, they must be extracted explicitly. Some of them include abbreviations, time spans and so on. The more of these patterns are learned by the system the better it can reason like humans. Some of these patterns are discussed in this paper. Again, we omit details due to lack of space.

Common Sense Knowledge Generation

The knowledge that we extract, represented using the predicates described in previous section, comes directly from the input text. We make use of additional knowledge sources such as the WordNet to gain supplementary information (common sense knowledge) about the concepts in the passage. CASPR uses the Hypernym relation from WordNet to build its ontology, coded as an answer set program.

To generate ontology rules, we use standard knowledge patterns from answer-set programming like the preference pattern and the default reasoning pattern (Gelfond and Kahl 2014; Chen et al. 2016). In general a concept is represented using the following signature throughout this paper:

concept(concept_instance, instance_sense)

For example: *e.g. lion(simba, noun_animal).*

Hypernym Processing: Hypernyms can be used to infer various properties of and functions about concepts. Hypernyms make use of the generalization principle to transfer properties from more general concepts to their specific concepts. There are three steps required to do so (i) Identify the concepts from the passage and generate a hypernym graph. (ii) Aggregate them to common base concepts. (iii) Generate hypernym rules. Figure 3 shows rules just generated for one of the senses of the concept "*lion*".

```
big_cat(X, noun_animal) :- lion(X, noun_animal)
feline(X, noun_animal) :- big_cat(X, noun_animal)
carnivore(X, noun_animal) :- feline(X, noun_animal)
placental(X, noun_animal) :- carnivore(X, noun_animal)
mammal(X, noun_animal) :- placental(X, noun_animal)
vertebrate(X, noun_animal) :- mammal(X, noun_animal)
chordate(X, noun_animal) :- vertebrate(X, noun_animal)
animal(X, noun_tops) :- chordate(X, noun_animal)
organism(X, noun_tops) :- animal(X, noun_tops)
living_thing(X, noun_tops) :- organism(X, noun_tops)
object(X, noun_tops) :- living_thing(X, noun_tops)
physical_entity(X, noun_tops) :- object(X, noun_tops)
entity(X, noun_tops) :- physical_entity(X, noun_tops)
```

Figure 3: Rules representing the animal branch of "lion"

Generating Hypernym Concept Graph: The first step in generating Hypernym rules is to generate a hypernym graph from the concepts found in the passage and create a concept bag containing all the hypernyms required for the passage. As hypernyms in WordNet also depend on the different senses of the word, we need to get hypernyms for all the different senses of the word as well. For example, the word "*car*" has 5 distinct senses.

The sense used for a motor vehicle is the most frequently used sense for the word "*car*". Now, as "vehicle" is a hypernym for this sense of "*car*" we can apply the properties of a "*vehicle*" like vehicle has capacity, it has mass, it is driven and many more to the "*car*". But apart from this meaning, vehicle also has three other meanings.

From all the meanings of vehicle only properties of the sense "*A conveyance to transport people*" can be applied to that of car in sense of a "*motor vehicle*". This makes considering the sense of the entity while extracting knowledge from WordNet crucial. Below is the algorithm mentioned to collect hypernyms of all the concepts in the text. In the algorithm, we go over all the senses of the word under consideration and add its hypernyms to the bag of concepts. We will then use this concept bag to generate the graph through aggregation as described in the next section.

Consider the word *lion*. The algorithm will generate the following four chains of relations corresponding to its four senses.

Sense 1: (animal) lion → big_cat → feline → carnivore → placental → mammal → vertebrate → chordate → animal → organism → living_thing → object → physical_entity → entity

Sense 2: (person) lion → celebrity → important_person → adult → person → organism → living_thing → object → physical_entity → entity

```
procedure GenerateHypernymOntology(word)
    if (word is a valid noun)
        // This method returns all the senses of the word from
        // WordNet ranked by frequency of use from most to least
        senses = GetAllRankedSenses(word)

        for each sense ∈ senses do
            GetHypernyms(sense, word)
            add sense to conceptBag
        end for
    end if
end procedure
```

Figure 4: Generating Hypernym Ontology Rules

Sense 3: (person) lion → person → organism → living_thing → object → physical_entity → entity

Sense 4: (location) lion → sign_of_the_zodiac → region → location → object → physical_entity → entity

Aggregation of Concepts: In this step we look at all the concepts in the concept bag and aggregate the concepts that have the same base word but have multiple senses under the same umbrella term. By doing this we can now represent a graph where all the concepts form nodes of the graph and different senses form the edges of the graph. The graph in Figure 5 is formed after merging the 4 senses mentioned above. Using this graph, we can directly generate the rules, for example, as shown in Figure 3 for lion.

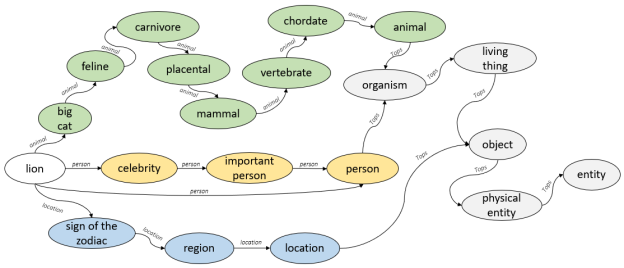

Figure 5: Concept Graph of Hypernym Relation

Adding Common Sense Knowledge Manually: Note that at the moment we are only using WordNet, however, other knowledge resources, such as YAGO (Mahdisoltani, Biega, and Suchanek 2015), VerbNet (Kipper et al. 2006), (Mitchell and others 2015), etc., can be incorporated as well to extract additional common sense knowledge. Doing so is relatively easy, as it mostly involves syntactic transformation to ASP syntax. Note that, currently, common sense knowledge about verbs is entered manually. An example of adding knowledge manually is the following:

Nikola_Tesla is similar to Tesla:

_similar(tesla, nikola_tesla).

A team X can represent an organization Y, if Y possess X:

event(E, represent, X, Y):-_possess(Y, X), organization(Y), team(X).

Note that knowledge that is added manually can reside in the knowledge-base permanently (very similar to humans,

where we learn knowledge once, and may use it later in another context).

Word Sense Disambiguation

Word sense disambiguation (WSD) is the task of selecting the best sense out of a collection of senses applicable to a concept. When queried from WordNet we get a list of senses for a specific concept ordered from the most used to the least used. WSD is a very common problem in NLP applications and has many statistical solutions that have been developed. We develop a method using negation as failure, classical negation and preferences that performs WSD correctly. Essentially, contextual knowledge is used to disambiguate the word sense, in a manner that humans use. We explain it next.

Representation of Word Senses: Word senses are represented using two different logic patterns in our research. Using both these patterns together, senses are selected for the various concepts in the text which activate their hypernym relations. The pattern discussed in this section, tries to prove a sense for a concept. Its template can be given as follows

c(X, s_i) :- c(X), properties_si(X), not -c(X, s_i).

In the above template, we are trying to prove that "*X* is an instance_of concept *c* with sense s_i". Here every concept *c* has one or more senses denoted by s_i. The above rule states that *X* is an instance of the concept *c* with the sense s_i only if we can prove that *X* is some instance of concept *c*, *X* has all the properties required to be of sense s_i, and we cannot prove that *X* is definitely *not* an instance of concept c with sense s_i. Such rules are generated for every sense s_i of the concept in the order of senses from the most used sense to the least used sense. The first term given by '*c(X)*' is responsible to short circuit and fail the rule if the instance *X* does not belong to the class *c*. The second term, '*properties_si(X)*', is the main predicate which tries to prove that *X* shows the properties of having sense s_i. It is this predicate that can be added either manually or using other sources to prove the sense. The third term of the body is a strong exception against the head of the rule, that fails the rule if an exception is found against the application of the sense.

According to WordNet, the concept '*tree*' has three senses, S = {*plant, diagram, person*}, ordered according to their frequency of use. Thus, using the above-mentioned template for the sense the three rules generated for the tree concept can be given as follows

tree (X, plant) :- tree (X), properties_plant (X),
not -tree (X, plant).
tree (X, diagram) :- tree (X), properties_diagram (X),
not -tree (X, diagram).
tree (X, person) :- tree (X), properties_person (X),
not -tree (X, person).

Preference Patterns for Senses: We humans perform word sense disambiguation in our day to day life by taking cues from the context. Over a period of time we learn which senses are more common than others and develop a preference order. Consider a concept *c* having three senses *s1*, *s2* and *s3* ordered according to the frequency of their use from the most used to the least used. Then, we first assume the sense to be *s1*, unless we know that *s1* is not the sense from some other source. Then we move on to the next sense *s2* unless we know that both *s1* and *s2* cannot be applicable. Then finally we choose *s3*. This process of elimination of senses and choosing senses according to preferences can be modeled as the following ASP code template.

c(X, s_p) :- c(X), not -c(X, s_p),
-c(X, s_1), -c (X, s_2), ..., -c (X, s_{p-1}),
not c (X, s_{p+1}), not c (X, s_{p+2}), ..., not c (X, s_n).

The above skeleton is applied for all the senses of concept c in order of preference. The above template represents the rule generated for the p^{th} sense of the concept *c* such that $1 < p < n$, where *n* is the total number of senses of concept *c*. If *p = 1* then the template omits the classical negation terms as follows

c(X, s_1) :- c(X), not -c(X, s_1), not c(X, s_2),
not c(X, s_3), ..., not c(X, s_n).

Similarly, if *p = n*, then the template omits the negation as failure terms, given as

c(X, s_n) :- c(X), not -c(X, s_n), -c(X, s_1),
-c(X, s_2), ... -c(X, s_{n-1}).

We can apply this pattern to the 'tree' concept as an example. A tree can be a living object (plant), a diagram, or a person (Mr. Tree).

tree (X, plant) :- tree (X), not -tree (X, plant),
not tree (X, diagram), not tree (X, person).
tree (X, diagram) :- tree (X), not -tree (X, diagram),
-tree (X, plant), not tree (X, person).
tree (X, person) :- tree (X), not -tree (X, person),
-tree (X, plant), -tree (X, diagram).

This pattern is responsible for assigning at least one sense for every concept in the text. This preference pattern along with the property pattern mentioned previously helps disambiguate word sense just as humans do.

Query Generation

Once knowledge has been generated from the text and auxiliary sources, our next task is to translate the question we want to answer into an ASP query.

Since a question is also a sentence, it is processed in the same way that any other sentence would as mentioned before in this paper. This means that a semantic graph is generated for every question and event regions are created within the question assigning event ids to different parts of the question. To generate an ASP query from the semantic graph the following steps are applied: (i) Question understanding (ii) Generation of query predicates (iii) Applying base constraints (iv) Combining constraints. Currently this module is only built to deal with simple interrogative sentences but can be extended to deal with more complex questions.

Question Understanding: For analyzing the question, we try to obtain four kinds of information from the question, namely, the question word, the question type as mentioned in the previous section, the answer word or the focus of the question and the answer type.

The question word is the Wh-word found in the question. In general, Wh-words can be found out by looking at the following POS Tags on the words: WDT, WP, WP$ and WRB. If none of the tags are found, then the questions

Table 1: Expected Answer Types for Question Types

Question Type	*Expected Answer Type*
WHERE	PLACE
WHO	PERSON
WHEN	TIME
HOW_MANY	NUMBER
HOW_MUCH	NUMBER
HOW_LONG	NUMBER [length]
HOW_FAR	NUMBER [distance]
WHAT	** variable **
WHICH	** variable **
UNKNOWN	UNKNOWN

may have a copula as its question word or a modal as its question word. The question type contains a set of predefined question types that help in further processing. These include WHAT, WHERE, WHO, WHICH, WHEN, HOW_MANY, HOW_MUCH, HOW LONG, HOW_FAR, and UNKNOWN. Once the main question word is found, we can use the word and its relations to determine the exact question type. The third type of information we try to extract is the answer word or the focus of the question. The answer word is the word in a question that tells us what kind of answer is expected from the question. Not all types of questions require an answer word, so in those cases this information is null. The fourth and the last information that we extract from the question is the answer type. The answer type depends on the question type. For most of the question types the required answer type is predefined. The answer types supported by the system are as follows SUBJECT, OBJECT, PLACE, PERSON, TIME, YEAR, DAY, MONTH, NUMBER and UNKNOWN. Table 1 shows expected answer types depending on question types. With the help of this basic analysis, we start generating the predicates for the query generation.

Generating Query Predicates: Using the information that we gathered during Question Understanding, we start generating predicate facts for all the words in the question. We will use predicates mentioned previously in the paper as a reference. There are some changes that need to be made in a few predicates, e.g., event predicate, property predicate, etc. The rest of the predicates like the possess, mod, and named entity predicates are generated in a manner similar to our earlier discussion.

Event Predicate: In case of questions, we do not know the event_id of the event in question so we replace it with a variable. The trigger_verb is in the question that triggers the predicate generation. The actor acts like the subject of the verb and the participant is the object or the modifier. The participants can be obtained from the direct object (*dobj*) relations of the verb. Let us now look at the various ways that we can obtain the actor or the subject in any event. Here, we consider three ways that help us find the subject in an event.

event (E_k, *verb*, S_k, O_k)
event (E_k, *verb*, _, _), *_relation* (S_k, E_k, *_clause*)
event (E_k, *verb*, _, O_k), *_property* (E_k, *verb*, *_by*, S_k)

While generating the predicates we use the verb id as the event or the verb index. The index *k* in the above predicates comes from the verb indices that are calculated from the semantic graph of the question. While generating these predicate patterns if we find out that the answer type is either a *SUBJECT* or an *OBJECT*, then we replace the subject (S_k) or the object tag (O_k) by the answer tag marked as 'X_k'.

Example 11. Given the question *"What company owns walt_disney?"*, the three possibilities for the event predicate are:

1. *event (E1, own, X1, O1), _similar (walt_disney, O1).*
2. *event (E1, own, _, O1), _property (E1, own, _by, X1), _similar (walt_disney, O1).*
3. *event (E1, own, _, _), _relation (X1, E1, _clause), _similar (walt_disney, O1).*

Property Predicate: Property predicates are generated from verbs and nouns that are modified by nominal phrases in the sentence. Here, like the event predicate, we are unaware of the event_id and hence it will be set as a variable. The modified_entity is the noun or the verb that triggered the predicate generation. The preposition here can either be obtained from the case relation or can be left blank (_). The modifier is the head of the nominal modifier that can be used to constraint the query. If the modifier is the answer word, then we replace the word with the answer tag (X_k).

Example 12. Given "On what streets is the ABC's headquarter located?", we produce
_property(E2, locate, on, X2).

Similar Predicate: The *similar* predicate models the concept of similarity between entities in which one entity is so like the other one that they can replace each other. The similar predicate thus models one of the principles of commonsense reasoning where we as humans make use of the similarity relationship while reasoning. For example, sometimes we use the last name of people to refer to them instead of their entire name, e.g., Einstein instead of Albert Einstein. Some rules for the similar predicate are as follows:

a. _similar (X, X).
b. _similar (X, Y) :- _abbreviation (X, Y).
c. _similar(X, Y) :- _abbreviation (Y, X).
d. _similar(X, Y) :- _is (X, Y).
e. _similar(X, Y) :- _similar(X, Z), _similar(Z, Y).

Generating Base Constraints: Base constraints are generated at the end after all the constraints from the question have been generated. These constraints refer to the constraint that depend on the answer type of the question. For the following cases we generate the base constraints as follows.

TIME → *time*(X_k)
DAY → *day*(T_k, X_k), *time*(T_k)
MONTH → *month*(T_k, X_k), *time*(T_k)
YEAR → *year*(T_k, X_k), *time*(T_k)
PLACE → *location* (X_k) or *location*(X_k, *noun_location*)
PERSON → *person* (X_k) or *person*(X_k, *noun_person*)

In case of the answer type being UNKNOWN, we may be expecting the answer to be a specific concept represented by the answer word. Thus, in such cases the base constraint comes from the answer word itself.

UNKNOWN $\rightarrow$ *concept(X_k)*
e.g., company(X_k) or city(X_k)

Combining constraints: After generation of the predicates from the question and the base constraints from answer type we combine all the constraints to create query. This can be best explained with an example.

Example 13. For the question "When was Nikola Tesla born?", the following queries will be generated and tried in order.

1. event (E2, bear, S2, O2), _similar (nikola_tesla, S2), property (E2, bear, on, X2), time(X2).
2. event (E2, bear, _, O2), _property (E2, bear, _by, S2), _similar (nikola_tesla, S2), property (E2, bear, on, X2), time(X2).
3. event (E2, bear, _, _), _relation (S2, E2, _clause), similar (nikola_tesla, S2), property (E2, bear, on, X2), time(X2).
4. _start_date (S2, X2), _similar (nikola_tesla, S2), time(X2).

In this sentence, we have the special predicates _start_date applying the constraints of timespans on an entity like 'Nikola Tesla'.

Query Confidence Classes: Ideally, we would want to be able to answer all questions with most number of constraints applied on the query so that we have a strong justification for the answer, but this is not always the case. In natural language the same questions can be posed in multiple different ways using synonymous concepts. This makes it more difficult to answer questions. Thus, it is a good idea to make any question answering system robust, so that it fails gracefully. We achieve this by introducing the concept of confidence classes on the generated queries and by relaxing constraints on the queries to make their execution more flexible. In the absence of an answer, the CASPR system starts removing constraints (sub-goals) in the query and relaxing it with the hope of executing it successfully and obtaining some answer. Currently, queries have been divided into 4 confidence classes that range from most confident (certain) to the least confident (guess):

Certain: These kinds of queries have all the constraints generated by our question processing module. If an answer is produced, it is a correct answer.

Likely: These kinds of queries drop all sub-goals that do not have variables.

Possible: These kinds of queries only contain the answer predicates and base constraints.

Guess: These queries only contain the base constraints.

As an example, suppose we ask the question: "When was Tesla born?", given a biographical passage about Tesla. If the query that is formulated succeeds, it will produce the certain answer (1856). However, if for some reason the query fails, in the worst case, we would just retrieve any year in the passage and report it as an answer. It should be noted that with enough knowledge, we are always guaranteed to produce the correct answer.

Note also that there are many other ways of relaxing constraints in the query to obtain less precise answers. The above is just one reasonable scheme.

Evaluation Results

The SQuAD Dataset (Rajpurkar et al. 2016) contains more than 100,000 reading comprehensions along with questions and answers for those reading passages. SQuAD dataset uses the top 500+ articles from the English Wikipedia. These articles are then divided into paragraphs. We used the Dev Set v1.1 of the SQuAD Dataset to obtain comprehension passages for building a prototype for the proposed approach. This dataset has around 48 different articles with each article having around 50 paragraphs each. Out of the 48 different articles in the SQuAD dev set, 20 articles were chosen from different domains to help build the CASPR system (these 20 passages and associated questions are uploaded on Easychair as a supplement). Using the 20 different articles mentioned above, the ASP program was generated on one paragraph from each article. Then, ASP queries were generated for all the questions in the dataset for these paragraphs. The results show the percentage of questions for which the answer generated from the ASP solver was present in the list of answers specified for the question in the SQuAD dataset. This result can be viewed in terms of the Table 2. The result shows that approximately 80% of the questions can be answered. This shows that most of the knowledge, if not all, has been captured successfully in the ASP program generated for the passage. The ASP queries generated for the questions are very similar to the original question and convey the same meaning.

Note that the two main reasons why a query may fail to produce an exact (certain) answer are: (i) the parse fails; and, (ii) lack of knowledge about the concepts in the question.

Example 14. The question *"What day was the game played on?"* will produce the two queries below:

1. event (E1, play, S1, O1), _similar (game, O1), _property (E1, play, on, T), day (T, X1), time (T).
2. event (E1, play, S1, O1), _property (E1, play, on, X1), time (X1).

The first query is more specific which asks for the exact day and the second one asks for the date or the time. Here both queries ask for "time on which the game or something synonymous to it was played". The above queries are semantically very close to the question asked. They produce the correct answer in our system.

Example 15. The question *"What city did Super Bowl 50 take place in?"* will produce the single query:

event (E1, _, S1, O1), _similar ('super_bowl_50', O1), _property (E1, _, in, X1), city(X1).

Here, we are querying for "Super Bowl 50 to take place in some city", which is semantically very close to the question under consideration. One of the main problem with question answering is that the question can be asked in lot of different forms. We as humans can fill in the gaps between the question and the passage, if any, with the help of similar meaning words and phrases using common sense knowledge, but the machine fails to do so due to absence of enough digital semantic resources. (Note that this query is also answered correctly by our system.)

Note that our system has shown excellent execution performance on the 171 questions on 20 passages tried thus far:

Table 2: Results for Question answering

No	Article	Result	%
1	ABC	5/5	100
2	Amazon Rainforest	12/14	85.7
3	Apollo	4/5	80
4	Chloroplasts	4/5	80
5	Computational Complexity	3/3	100
6	Ctenophora	9/12	75
7	European Union Law	13/13	100
8	Genghis Khan	3/5	60
9	Geology	4/5	80
10	Immune System	13/15	86.67
11	Kenya	5/5	100
12	Martin Luther	2/5	40
13	Nikola Tesla	6/7	85.7
14	Normans	4/5	80
15	Oxygen	8/15	53.3
16	Rhine	5/8	62.5
17	Southern California	3/5	60
18	Steam Engine	4/5	80
19	Super Bowl 50	25/29	86.2
20	Warsaw	3/5	60
Total		135/171	78.95
Average Result		77.76%	

80% of the questions were answered in 2 to 3 milliseconds, while the rest were answered in a few seconds.

Contributions and Related Work

The main contribution of this paper is an effective and efficient method for converting textual data into knowledge represented as an answer set program. This includes developing a neo-davidsonian logic inspired generic calculus that help represent knowledge, and using knowledge sources such as WordNet to acquire common sense knowledge about terms found in the text to create a custom ontology for the problem at hand. This helps in ensuring that our system is scalable as well as shows good execution performance as this custom ontology can be generated dynamically. Yet another novelty is in showing how word sense disambiguation can be elegantly modeled using answer set programming.

The paper also proposes a framework for converting natural language questions into ASP queries. These queries can be run on an ASP system such as s(ASP) to compute answers. The query generation framework is made robust through broadening of queries by dropping constraints, thus increasing the possibility that the question will indeed be answered (even though in the worst case the answer may just be a guess). This approach to handling question answering is yet another novelty of the proposed system.

Wrt related work, Cyc (Wikipedia contributors 2018) is one of the oldest AI project that attempts to model common sense reasoning. In Cyc, knowledge is presented in the form of a vast collection of ontologies that consist of implicit knowledge and rules about the world that represents common sense knowledge. Cyc uses a *community of agents* consisting of multiple reasoning agents that rely on more than 1000 heuristic modules to solve inference problems. Cyc, however, does not work with a natural language, though recently some efforts have been started. A potential problem with Cyc is knowing which agent to apply, and which heuristic to use. For a common sense reasoning system to be successful, it has to be modeled in a very simple way. Otherwise, we may possess the individual pieces of knowledge to answer a given question, but may not be able to compose these pieces together to arrive at an answer. For this reason, in CASPR, knowledge is represented using very few generic predicates, and simple ASP-based reasoning patterns are used to compute answers. Our initial experiments suggest that our approach is effective.

A similar approach has been taken by Bos (Bos 2009), where he imports knowledge from WordNet. Bos uses full first theorem proving, making the modeling of common sense reasoning complex. Vo and Baral (Vo, Mitra, and Baral 2015) have developed the NL2KR tool that allows natural language text to be translated to an answer set program. Several researchers have worked on applying ASP for NLP tasks and many of these efforts are reported in the first workshop on NLP and Automated Reasoning (Baral and Schüller 2013). Our approach has many elements common with these efforts, however, our work is based on the query-driven s(ASP) predicate ASP engine, and thus is scalable and not constrained by limitations of grounding based implementations of ASP. We omit details of the comparison with other efforts due to lack of space.

There are many approaches to question answering based on machine learning (cf. SQuAD website). However, they are not based on "text understanding" and so can only answer questions related to data they are trained on.

Conclusions

In this paper we presented a comprehensive natural language question answering system called CASPR. The system translates natural language text and questions into ASP code and ASP query, respectively. It also imports common sense knowledge from other sources such as WordNet. We also presented an elegant solution for word sense disambiguation based on default and classical negation. CASPR has shown promising results on passages taken from the Stanford SQuAD natural language dataset.

A critical component of CASPR's success is the s(ASP) query-driven, predicate ASP system. The s(ASP) system's query driven nature results in three major advantages for CASPR: (i) only parts of the knowledge base relevant to answering the question are explored during execution; (ii) justification for answers can be extracted from the justification tree produced by s(ASP); and, (iii) the question answering system is scalable, as no grounding of the program needs to be done as s(ASP) can execute predicates directly under the stable model semantics.

Future work includes (i) extending the system to handle more complex questions including causality questions, (ii) incorporating additional knowledge resources for importing more common sense knowledge, (iii) Generating justifications to a question's answer in a more human-readable way.

Acknowledgement

Research partially supported by NSF Grant IIS 1718945. We are grateful to Vincent Ng, Elmer Salazar, Zhuo Chen, Farhad Shakerin, Sarat Varanasi, Kyle Marple, Joaquin Arias, and Takshak Desai for discussions.

Note: The 20 passages used and their associated questions are reproduced in an appendix uploaded as supplementary material on Easychair, if the reviewer wishes to view them.

References

Alviano, M.; Faber, W.; Leone, N.; Perri, S.; Pfeifer, G.; and Terracina, G. 2011. The disjunctive datalog system dlv. In *Datalog Reloaded*. Springer. 282–301.

Arias, J.; Carro, M.; Salazar, E.; Marple, K.; and Gupta, G. 2018. Constraint answer set programming without grounding. *arXiv preprint arXiv:1804.11162*.

Baral, C., and Schüller, P., eds. 2013. *Proceedings of the 1st Workshop on Natural Language Processing and Automated Reasoning 2013*, volume 1044 of *CEUR Workshop Proceedings*. CEUR-WS.org.

Baral, C. 2003. *Knowledge representation, reasoning and declarative problem solving*. Cambridge university press.

Bos, J. 2009. Applying automated deduction to natural language understanding. *J. Applied Logic* 7(1):100–112.

Chen, D., and Manning, C. 2014. A fast and accurate dependency parser using neural networks. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 740–750.

Chen, Z.; Marple, K.; Salazar, E.; Gupta, G.; and Tamil, L. 2016. A physician advisory system for chronic heart failure management based on knowledge patterns. *Theory and Practice of Logic Programming* 16(5-6):604–618.

Davidson, D. 1984. *Inquiries into Truth and Interpretation*. Oxford University Press.

De Marneffe, M.-C., and Manning, C. D. 2008. Stanford typed dependencies manual. Technical report, Technical report, Stanford University.

De Marneffe, M.-C.; Dozat, T.; Silveira, N.; Haverinen, K.; Ginter, F.; Nivre, J.; and Manning, C. D. 2014. Universal stanford dependencies: A cross-linguistic typology. In *LREC*, volume 14, 4585–4592.

Finkel, J. R.; Grenager, T.; and Manning, C. 2005. Incorporating non-local information into information extraction systems by gibbs sampling. In *Proceedings of the 43rd annual meeting on association for computational linguistics*, 363–370. Association for Computational Linguistics.

Gebser, M.; Kaminski, R.; Kaufmann, B.; Ostrowski, M.; Schaub, T.; and Thiele, S. 2010. gringo, clasp, clingo, and iclingo. *user guide*.

Gelfond, M., and Kahl, Y. 2014. *Knowledge representation, reasoning, and the design of intelligent agents: The answer-set programming approach*. Cambridge University Press.

Jonson-Laird, P. 2009. *How We Reason*. Oxford University Press.

Kipper, K.; Korhonen, A.; Ryant, N.; and Palmer, M. 2006. Extending verbnet with novel verb classes. In *Proceedings of the Fifth International Conference on Language Resources and Evaluation, LREC 2006, Genoa, Italy, May 22-28, 2006.*, 1027–1032.

Kowalski, R., and Sergot, M. 1989. A logic-based calculus of events. In *Foundations of knowledge base management*. Springer. 23–55.

Mahdisoltani, F.; Biega, J.; and Suchanek, F. M. 2015. YAGO3: A knowledge base from multilingual wikipedias. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*.

Marple, K.; Salazar, E.; and Gupta, G. 2017. Computing stable models of normal logic programs without grounding. *arXiv preprint arXiv:1709.00501*.

Miller, G. A. 1995. Wordnet: a lexical database for english. *Communications of the ACM* 38(11):39–41.

Mitchell, T. M., et al. 2015. Never-ending language learning. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA.*, 2302–2310.

Pendharkar, D. 2018. *An Answer Set Programming based Approach to Representing and Querying Textual Knowledge*. M.S. Thesis, The University of Texas at Dallas, `http://utdallas.edu/~gupta/dpthesis.pdf`.

Rajpurkar, P.; Zhang, J.; Lopyrev, K.; and Liang, P. 2016. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*.

Toutanova, K., and Manning, C. D. 2000. Enriching the knowledge sources used in a maximum entropy part-of-speech tagger. In *Proceedings of the 2000 Joint SIGDAT conference on Empirical methods in natural language processing and very large corpora*, 63–70. Association for Computational Linguistics.

Toutanova, K.; Klein, D.; Manning, C. D.; and Singer, Y. 2003. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*, 173–180. Association for Computational Linguistics.

Van Harmelen, F.; Lifschitz, V.; and Porter, B. 2008. *Handbook of knowledge representation*, volume 1. Elsevier.

Vo, N. H.; Mitra, A.; and Baral, C. 2015. The NL2KR platform for building natural language translation systems. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, ACL 2015, July 26-31, 2015, Beijing, China, Volume 1: Long Papers*, 899–908.

Wikipedia contributors. 2018. Cyc — Wikipedia, the free encyclopedia. [Online; accessed 17-May-2018].